

Изучаем Angular 2

Пабло Дилеман

[PACKT] PUBLISHING

ДМК издательство

Пабло Дилеман

Изучаем Angular 2

Pablo Deeleman

Learning Angular 2

Your quick, no-nonsense guide to building real-world apps with Angular 2

[PACKT] PUBLISHING
BIRMINGHAM – MUMBAI

open source
community experience distilled

Пабло Дилеман

Изучаем Angular 2

Краткое практическое руководство по созданию приложений с помощью Angular 2

Москва, 2017

УДК 004.738.5:004.4AngularJS
ББК 32.973.26-018.2
Д46

Дилеман П.
Д46 Изучаем Angular 2 / пер. с англ. Р. Н. Рагимова; под науч. ред. А. Н. Киселева. – М.: ДМК Пресс, 2017. – 354 с.: ил.

ISBN 978-5-97060-461-8

Фреймворк Angular 2 полностью переписан с нуля для удовлетворения требований современных разработчиков, которым необходима высокая производительность и отзывчивость веб-приложений.

Эта книга поможет вам освоить основы проектирования и разработки компонентов Angular 2, обеспечивая полный охват синтаксиса языка TypeScript. После создания первых компонентов мы объединим их в полноценное веб-приложение. Затем перейдем к реализации маршрутов в Angular 2, узнаем, как организовать поддержку разных состояний приложения и навигацию между компонентами. После этого мы займемся веб-формами и проверкой ввода пользователя, а затем используем новые знания как фундамент для изучения основ аутентификации пользователей и различных способов защиты страниц и разделов сайта. Также в этой книге рассматривается тема анимации компонентов и элементов DOM. В заключительной части дается детальное освещение приемов модульного тестирования, таких как службы, директивы, маршруты и фильтры.

Издание предназначено веб-разработчикам, которые хотят освоить разработку современных мобильных и настольных веб-приложений следующего поколения с помощью Angular 2.

УДК 004.738.5:004.4AngularJS
ББК 32.973.26-018.2

Copyright ©Packt Publishing 2016. First published in the English language under the title “Learning Angular 2” – (9781785882074).

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1-78588-207-4 (анг.)
ISBN 978-5-97060-461-8 (рус.)

Copyright © 2016 Packt Publishing
© Оформление, издание, перевод, ДМК Пресс, 2017

Посвящаю эту книгу моим родителям Паулу и Пепе
и памяти моего любимого брата Хосе Рауля.

Ты будешь вечно жить в наших сердцах.

Содержание

Об авторе 13

Благодарности 14

О технических рецензентах 15

Предисловие 16

Глава 1. Создание самого первого компонента в Angular 2 22

Обновленный подход 23
Веб-компоненты 24
Почему именно TypeScript? 25
Настройка рабочего окружения 26
Установка зависимостей 26
Установка TypeScript 30
Установка утилиты typings 31
Hello, Angular 2! 33
Классы TypeScript 34
Введение в декораторы 34
Компиляция TypeScript в понятный браузерам код на JavaScript 35
HTML-контейнер 36
Сопроводительные примеры книги 39
Соединяем все вместе 40
Усовершенствование интегрированной среды разработки 43
Sublime Text 3 43
Atom 44
Visual Studio Code 45
WebStorm 46
Использование Gulp с другими интегрированными средами разработки 46
Погружение в компоненты Angular 2 48
Улучшение производительности 48
Методы компонента и обновление данных 48
Добавление интерактивности в компонент 51

Улучшения для вывода данных в представлении и доработка пользовательского интерфейса ... 53
Итоги ... 57

Глава 2. Введение в TypeScript ... 58
Причины создания TypeScript ... 58
Преимущества TypeScript ... 59
Знакомство с ресурсами, посвященными TypeScript ... 61
Типы в TypeScript ... 62
Тип string ... 63
Тип number ... 64
Динамическая типизация с помощью типа any ... 64
Тип void ... 66
Автоматическое определение типов ... 66
Функции, лямбда-функции и поток выполнения ... 67
Аннотации типов в функциях ... 67
Параметры функций в TypeScript ... 69
Улучшенный синтаксис функций и область обработки с лямбда-функциями ... 71
Классы, интерфейсы и наследование классов ... 73
Анатомия классов: конструкторы, свойства и методы ... 73
Интерфейсы в TypeScript ... 76
Расширение классов с помощью наследования ... 78
Декораторы TypeScript ... 79
Декораторы классов ... 80
Декораторы свойств ... 81
Декораторы методов ... 83
Декораторы параметров ... 86
Работа с модулями в приложении ... 87
Внутренние модули ... 87
Внешние модули ... 89
Предстоящий путь ... 90
Итоги ... 90

Глава 3. Реализация свойств и событий компонентов ... 92
Улучшенный синтаксис шаблонов ... 92
Привязка данных с помощью входных свойств ... 93
Дополнительные синтаксические конструкции для привязки выражений ... 94

Привязка событий к выходным свойствам 94
Входные и выходные свойства в действии 95
Передача данных в пользовательских событиях 100
Локальные ссылки в шаблонах 102
Альтернативный синтаксис для входных и выходных свойств 103
Настройка шаблона класса компонента 104
Внутренние и внешние шаблоны 104
Инкапсуляции CSS-стилей 105
Управление инкапсуляцией представлений 107
Итоги 108

Глава 4. Расширение компонентов с помощью директив и фильтров 110

Директивы в Angular 2 111
Встроенные директивы 111
Управление привязками шаблонов с помощью фильтров 114
Фильтры uppercase/lowercase 115
Фильтры number, percent и currency 115
Фильтр slice 117
Фильтр date 117
Фильтр JSON 118
Фильтр replace 118
Фильтры интернационализации 119
Фильтр async 120
Реализация списка заданий в проекте Pomodoro 120
Создание основного HTML-контейнера 121
Построение таблицы со списком заданий с помощью директив Angular 122
Переключение заданий в списке 128
Отображение изменений состояния в шаблонах 130
Встраивание дочерних компонентов 132
Создание собственных фильтров 136
Анатомия пользовательских фильтров 136
Пользовательские фильтры для форматирования времени 137
Фильтрация данных с помощью пользовательских фильтров 139
Создание собственных директив 140
Анатомия пользовательских директив 141
Создание подсказки для заданий с помощью пользовательской директивы 143

Несколько слов об именах пользовательских директив и фильтров....145
Итоги....146

Глава 5. Создание приложений с помощью компонентов Angular 2....147

Введение в деревья компонентов....148
Общие соглашения для масштабируемых приложений....149
Соглашение об именовании файлов и модулей....151
Обеспечение непрерывной масштабируемости с помощью фасадов или баррелей....152
Внедрение зависимостей в Angular 2....153
Внедрение зависимостей в дереве компонентов....157
Переопределение поставщиков в иерархии инструментов внедрения зависимостей....161
Расширенная поддержка инструментов внедрения зависимостей для пользовательских сущностей....163
Инициализация приложения с помощью функции bootstrap()....165
Подготовка структуры каталогов приложения Pomodoro....167
Приведение приложения в соответствие с рекомендациями Angular 2....170
Общий контекст....170
Создание фасадного модуля, включающего пользовательскую баррель поставщиков....175
Создание компонентов....177
Начальная загрузка приложения....186
Итоги....187

Глава 6. Асинхронная обработка данных в Angular 2....188

Способы асинхронной обработки информации....189
Наблюдаемые объекты....191
Реактивное функциональное программирование в Angular 2....193
Библиотека RxJS....196
Введение в HTTP API....199
В каких случаях используются классы Request и RequestOptionsArgs....200
Объект Response....201
Обработка ошибок при выполнении Http-запросов....202
Внедрение класса Http и псевдоним HTTP_PROVIDERS....202

Пример: отслеживание данных по HTTP 204
Итоги 209

Глава 7. Маршрутизация в Angular 2 210
Добавление поддержки маршрутизатора Angular 2 211
Настройка службы маршрутизации 212
Создание нового компонента для демонстрационных целей 214
Настройка декоратора RouteConfig с помощью экземпляров RouteDefinition 216
Директивы маршрутизатора RouterOutlet и RouterLink 218
Принудительный переход по маршрутам 220
CSS-ловушки для активных маршрутов 222
Параметры обработки маршрута 222
Передача динамических параметров в маршрутах 223
Разбор параметров маршрута с помощью службы RouteParams 225
Определение дочерних маршрутизаторов 227
События жизненного цикла маршрутизатора 230
Событие CanActivate 231
Событие OnActivate 233
События CanDeactivate и OnDeactivate 234
События CanReuse и OnReuse 235
Переадресация на другие маршруты 237
Точная настройка базового пути 238
Настройка генерируемых URL-адресов с учетом стратегии размещения 239
Асинхронная загрузка компонентов с помощью AsyncRoutes 240
Итоги 241

Глава 8. Формы и аутентификация в Angular 2 243
Двухсторонняя привязка данных в Angular 2 244
Директива NgModel 245
Типы привязки к форме с помощью директивы NgModel 247
Отслеживание изменений и проверка допустимости данных 251
Элементы управления, группы элементов управления и класс FormBuilder 255
Объекты Control и Validator 256
Элементы управления в модели DOM и директива ngControl 257
Группировка элементов в модели DOM с помощью директивы NgControlGroup 258

Императивное определение групп элементов управления с помощью ControlGroup 260
Соединение модели DOM и контроллера с помощью ngFormModel 262
Пример компонента входа пользователя 263
Контекст функции входа 263
Шаблон формы входа 266
Компонент входа 266
Применение пользовательских проверок допустимости к элементам управления 268
Отображение изменений состояния в элементах управления 270
Фиктивная служба проверки подлинности клиента 271
Блокирование несанкционированного доступа 277
Реакция пользовательского интерфейса на состояние аутентификации пользователя 279
Улучшение управления доступом 281
Разработка собственной директивы безопасности RouterOutlet 282
Итоги 288

Глава 9. Анимация компонентов в Angular 2 289
Анимация с помощью стандартной CSS-разметки 290
Анимация с помощью классов-событий CSS 293
Анимация компонентов с помощью класса AnimationBuilder 295
Отслеживание состояния анимации с помощью класса Animation 300
Разработка пользовательских директив анимации 301
Взгляд в будущее: ngAnimate 2.0 307
Итоги 308

Глава 10. Модульное тестирование в Angular 2 309
Почему тестирование необходимо? 310
Средства модульного тестирования в Angular 2 311
Внедрение зависимостей в модульных тестах 312
Настройка среды тестирования 315
Реализация средства выполнения тестов 315
Настройка команд NPM 318
Пользовательские функции сопоставления Angular 2 319
Тестирование фильтров 320
Тестирование компонентов 322

Переопределение зависимостей компонентов для улучшения тестирования 330
Тестирование маршрутов 332
Тестирование переадресации 334
Тестирование служб 334
Реализация фиктивных Http-ответов с помощью MockBackend 338
Тестирование директив 342
Предстоящий путь 343
Использование Jasmine в сочетании с Karma 344
Получение отчетов об охвате кода тестами 344
Реализация E2E-тестов 345
Итоги 345

Предметный указатель 347

Об авторе

Пабло Дилеман (Pablo Deeleman) в 90-х годах работал дизайнером пользовательских интерфейсов, когда открыл для себя Интернет. В то время модем со скоростью 14 400 бит/с был ключом в волшебный мир, а самой популярной была игра с названием «Создай свой сайт».

После получения степени бакалавра (с отличием) в области маркетинга и поработав в рекламной индустрии, он остался верен себе и погрузился в самостоятельное изучение, превратившись в увлеченного дизайнера пользовательских интерфейсов и разработчика веб-приложений с блестяще подготовленными CSS-макетами и толстыми клиентами на JavaScript. Пабло создал бесчисленное множество интерактивных проектов и веб-приложений для настольных и мобильных устройств.

В течение нескольких лет он сделал карьеру дизайнера и разработчика, успешно руководившего интернет-проектами для широкого круга клиентов. Работал с европейскими туроператорами, начинающими компаниями из Кремниевой долины, международными веб-сайтами с тяжелым интернет-трафиком, порталами глобальных банковских систем, компаниями, занимающимися созданием игровых программ для мобильных устройств, и это далеко не полный список. Появление Node.js и фреймворков одностраничных приложений стало поворотным пунктом в его карьере, заставив сосредоточиться на создании взаимодействий с пользователем, управляемых с помощью JavaScript.

Пожив и поработав в нескольких странах, Пабло Дилеман обосновался в Барселоне (Испания) и теперь возглавляет разработку интерфейсов в Барселонской студии компании Gameloft, мирового лидера в области разработки игровых приложений для мобильных устройств, известного такими играми, как «Despicable Me: Minions Rush» и «Asphalt 8».

Большую часть свободного времени, когда он не пишет книги и не принимает участия в корпоративных мероприятиях и переговорах, Пабло посвящает своему увлечению – игре на фортепиано и гитаре.

Благодарности

Книга, которую вы сейчас держите в руках, является результатом больших усилий и массы затраченного на нее времени. Однажды кто-то мудро заметил, что писать книгу о фреймворке, находящемся в стадии альфа-версии, – это то же самое, что стрелять навскидку по движущейся мишени, и это действительно так. В процессе написания книги автор и команда проекта сбились со счета, сколько раз пришлось ее переписывать, чтобы привести в соответствие с самой последней реализацией фреймворка. Немногие способны выдержать накал такого сражения, не разочароваться и не согласться с мыслью о прекращении проекта. Поэтому я хочу выразить благодарность команде издательства Packt и прежде всего Саманте Гонсалвес (Samantha Gonsalves). Теплые слова их поддержки подпитывали меня энергией, так необходимой для продвижения проекта.

Я также хотел бы особо поблагодарить своего друга Хорхе Феррандо (Jorge Ferrando) за помощь и подсказки в процессе работы над книгой. Его опыт работы с Angular 2 был бесценен при выборе между разными видами подачи обучающего материала. Кроме того, следует упомянуть моих коллег-разработчиков: Хавьера Гомеса (Javier Gómez), Альфонсо Фернандеса (Alfonso Fernández), Франа Ириэла (Fran Iruela) и Педро Нарциco (Pedro Narciso).

Еще я хотел бы поблагодарить всех моих наставников, направлявших мою профессиональную деятельность на протяжении нескольких лет, и особенно сотрудников компаний Casumo и Gameloft: Расмуса Свеннингсона (Razmus Svenningson), Кима Ларсена (Kim Larsen), Джосефа Галеа (Josef Galea), Стива Аттарда (Steve Attard), Идена Аззопарди (Iden Azzopardi), Ренальда Далли (Renald Dalli), Мэтью Борга (Matthew Borg), Марка Бусуттила (Mark Busuttil), Жерара Гинэ (Gerard Giné), Антонио Гонсалеса (Antonio González), Альберта Пуэртоласа (Albert Puértolas), Рафаэля Марфил (Rafael Marfil) и всегда вдохновляющего меня Стюарда Лангриджа (Stuart Langridge).

О технических рецензентах

Йоханнес Вебер (Johannes Weber) – увлеченный разработчик и консультант в области применения веб-технологий в промышленных JS-приложениях. Работает в компании Mayflower GmbH (Мюнхен, Германия), где специализируется на миграциях SPA и MPA. В свободное время (со)организует Мюнхенские конференции AngularCamp и JS-Kongress.de по AngularJS. Йоханнес сотрудничает с сайтом ESnextNews.com, который готов еженедельно присылать вам по электронной почте пять отличных ссылок по теме ECMAScript.next.

Предисловие

За последние годы Angular 1.x стал одним из наиболее популярных фреймворков на JavaScript, предназначенных для создания современных веб-приложений, больших и малых. Но по мере увеличения размеров и сложности приложений все явственнее проявлялись его недостатки в отношении производительности и масштабируемости. Angular 2 был полностью переписан для удовлетворения требований разработчиков современных веб-приложений, которым необходимы высокая производительность и гибкость.

Фреймворк Angular 2 разработан в соответствии с современными веб-стандартами и предоставляет определенную свободу в выборе языка, обеспечивая полную поддержку не только ES6 и TypeScript, но и широко распространенных сегодня ES5, Dart и CoffeeScript. Его встроенный механизм внедрения зависимостей позволяет создавать масштабируемые модульные приложения с ясным и выразительным кодом, значительно упрощая сопровождение и разработку, основанную на тестировании. Везде, где бы не применялся Angular 2, он демонстрирует беспрецедентный уровень быстродействия и производительности благодаря новой системе обнаружения изменений, которая работает в пять раз быстрее, чем ее предыдущая реализация. Более прозрачные представления и непревзойденный синтаксис стандартных шаблонов предоставляют бесконечный список мощных функций для создания мобильных и настольных веб-приложений нового поколения.

Фреймворк Angular 2 сохранит и упрочит свою роль в разработке современных веб-приложений на годы вперед. Но именно из-за его архитектуры, разрушающей стереотипы, освоение Angular 2 может стать непосильной задачей для новичков.

И здесь на помощь придет эта книга, цель которой – помочь читателю избежать желудочных колик при изучении документации с описанием фреймворка и его программного интерфейса, предложив практически подход к освоению фреймворка, позволяющий сразу перейти к его использованию. И это будет сделано с самого ее начала.

Книга описывает процесс построения веб-проекта на новой платформе, основанной на синтаксисе TypeScript, начиная с основных понятий, примеров компонентов и их использования для создания

функционала, усложняющегося с каждой главой, пока в конце книги не будет получено завершенное, протестированное и готовое к эксплуатации веб-приложение.

Какие темы охватывает книга

Глава 1 «Создание самого первого компонента в Angular 2» знакомит с веб-компонентами – строительными блоками всех приложений, реализованных с помощью Angular 2.

Глава 2 «Введение в TypeScript» знакомит с синтаксисом и особенностями типизированного надмножества ECMAScript 6, выбранного командой Angular для создания Angular 2.

Глава 3 «Реализация свойств и событий компонентов» описывает поведение компонентов, которые изменяют свое состояние в соответствии с данными, полученными через входные свойства, и передают данные в виде событий через выходные свойства.

Глава 4 «Расширение компонентов с помощью директив и фильтров» содержит полное описание фильтров, используемых для обработки данных в шаблонах, а также встроенных директив, расширяющих возможности компонентов. В этой главе читатель также узнает, как создавать собственные фильтры и директивы

Глава 5 «Создание приложений с помощью компонентов Angular 2» включает повторение изученных ранее элементов и показывает, как обеспечить произвольное масштабирование проектов на Angular 2 и выполнение соглашений об именовании и оформлении кода.

Глава 6 «Асинхронная обработка данных в Angular 2» рассматривает внедрение и развертывание HTTP-соединений со сторонними службами с помощью модуля Http для создания собственных клиентов.

Глава 7 «Маршрутизация в Angular 2» знакомит с механизмом маршрутизации в Angular 2 и его встроенными директивами, обеспечивающими различные способы загрузки компонентов посредством маршрутов и обработку состояний через программный интерфейс History API.

Глава 8 «Формы и аутентификация в Angular 2» демонстрирует различные способы создания веб-форм с двухсторонней приязкой данных к элементам ввода, а также создание сложных форм и проверку их допустимости.

Глава 9 «Анимация компонентов в Angular 2» рассматривает инструменты и классы для реализации анимационных эффектов, начиная с использования только стилей CSS, управляемых директивами

Angular 2, и заканчивая сложными переходами, управляемыми из JavaScript через компоновщики анимации Angular 2.

Глава 10 «Модульное тестирование в Angular 2» содержит пошаговое описание приемов тестирования приложений и типичных шаблонов развертывания модульных тестов для компонентов, директив, фильтров, маршрутов и служб.

Что потребуется для работы с книгой

Для работы с примерами в этой книге прежде всего понадобится веб-браузер, обновленный до последней версии. Мы рекомендуем Google Chrome или Mozilla Firefox, хотя Angular 2 поддерживает все современные браузеры.

Также понадобится программа-терминал, так как многие операции будут выполняться через консольные команды npm, для запуска которых также понадобятся установленные в системе Node.js и npm. Прочие необходимые модули и процедуры и порядок их установки будут описаны по ходу работы.

И наконец, для программирования вам потребуется текстовый редактор. В *главе 1 «Создание самого первого компонента в Angular 2»* приводится подробное описание лучших на настоящий момент интегрированных сред разработки приложений, использующих фреймворк Angular 2.

Кому адресована эта книга

Эта книга адресована веб-разработчикам, которые хотят освоить разработку современных мобильных и настольных веб-приложений следующего поколения с помощью Angular 2. От читателя не требуется опыт работы с Angular 1.x или 2, но предполагается хорошее владение JavaScript. Она отлично подойдет новичкам в Angular, нуждающимся в пояснении и определении его концепций.

Соглашения

В этой книге используется несколько разных стилей оформления текста для выделения разных видов информации. Ниже приведены примеры этих стилей с объяснением их назначения.

Программный код в тексте, имена таблиц баз данных, имена папок, имена файлов, расширения файлов, фиктивные адреса URL, пользовательский ввод и ссылки в Twitter будут выглядеть так: «В

результате этого действия в корневой папке проекта появится новый файл `tsconfig.json`, содержащий параметры, необходимые компилятору TypeScript для преобразования кода компонента в обычный код на JavaScript спецификации ECMAScript 5, который способны обработать современные браузеры».

Блоки программного кода оформляются так:

```
<body>
  <nav class="navbar navbar-default navbar-static-top">
    <div class="container">
      <div class="navbar-header">
        <strong class="navbar-brand">My Pomodoro Timer</strong>
      </div>
    </div>
  </nav>
  <pomodoro-timer></pomodoro-timer>
</body>
```

Когда потребуется привлечь ваше внимание к определенному фрагменту в блоке программного кода, он будет выделяться жирным:

```
<body>
  <nav class="navbar navbar-default navbar-static-top">
    <div class="container">
      <div class="navbar-header">
        <strong class="navbar-brand">My Pomodoro Timer</strong>
      </div>
    </div>
  </nav>
  <pomodoro-timer></pomodoro-timer>
</body>
```

Ввод и вывод в командной строке будут оформляться так:

```
$ npm install angular2 es6-shim es6-promise reflect-metadata rxjs zone.js
--save
```

Новые термины и **важные слова** будут выделены жирным. Текст, отображаемый на экране, например в меню или в диалогах, будет оформляться так: «Раздел **learn** предоставляет доступ к обучающим материалам, позволяющим ускорить освоение языка».



Предупреждения и важные сообщения будут выделены подобным образом.



Подсказки и советы будут выглядеть так.

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Поддержка пользователей

Если вы гордый владелец книги, изданной в Packt, мы готовы предоставить вам дополнительные услуги, чтобы ваша покупка принесла вам максимальную пользу.

Загрузка исходного кода примеров

Вы сможете загрузить файлы с примерами кода на странице:

https://github.com/deeleman/learning-angular2.

Кроме того, скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com или www.дмк.рф в разделе «Читателям – Файлы к книгам».

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки всё равно случаются. Если вы найдёте ошибку в одной из наших книг – возможно, ошибку в тексте или в коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии этой книги.

Если вы найдёте какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в Интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Packt очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в Интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли принять меры.

Пожалуйста, свяжитесь с нами по адресу электронной почты dmkpress@gmail.com со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Вопросы

Вы можете присылать любые вопросы, касающиеся данной книги, по адресу dm@dmk-press.ru или questions@packtpub.com. Мы постараемся разрешить возникшие проблемы.

Глава 1

Создание самого первого компонента в Angular 2

Если проследить тенденции за последнюю пару лет, можно отметить всплеск появления современных веб-фреймворков и библиотек на JavaScript для создания интерфейсов. Уже достигнута стадия, когда каждый день рождаются новые фреймворки, что заставляет разработчиков тщательно оценивать их, чтобы определить, стоит ли тратит время и силы на их освоение, а также их пригодность для применения в дальнейших проектах.

В конце концов, некоторые из них признаются наиболее актуальными. Вполне возможно, что следующие названия вам уже известны: Backbone, Ember, Knockout, Angular 1 и т. д.

В битву за господство в мире JavaScript вступили такие новые фреймворки, как React или Aurelia, отдающие предпочтение веб-компонентам и теневой модели DOM, которые являются краеугольным камнем их архитектуры. Приложения, построенные на их основе, получаются более модульными, масштабируемыми и простыми в сопровождении, не говоря уже об их непревзойденном уровне производительности.

С момента создания фреймворка Angular 1 прошло достаточно времени, чтобы его недостатки стали очевидными и с ними нельзя уже было мириться. Пришло время, когда отдельных улучшений в коде оказалось недостаточно. Потребовалось полностью сменить сам подход. В результате появился совершенно новый фреймворк, Angular 2, разработанный с нуля и полностью отвечающий новейшим тенден-

циям в отрасли. Центральное место в нем занимают веб-компоненты, а мощь теневой модели DOM используется для уменьшения времени реакции веб-сущностей на изменение состояния. Кроме того, Angular 2 включает систему обнаружения изменений состояния, встроенную в каждый компонент и отвечающую за организацию связей во всем дереве компонентов, составляющих приложение.

Определяющие черты Angular 2 выводят его за рамки простого фреймворка, основанного на веб-компонентах, поскольку его возможности охватывают практически все необходимое для построения современных веб-приложений: взаимодействие компонентов, универсальную поддержку различных платформ и устройств, превосходную технологию внедрения зависимостей, гибкий расширенный механизм маршрутизации для ослабления связей между компонентами, расширенную поддержку HTTP-сообщений, анимацию и интернационализацию. И это далеко не полный список.

В этой главе мы:

- узнаем причины уникальности Angular 2 в сравнении с предыдущими версиями;
- настроим среду для работы с Angular 2 и TypeScript;
- выберем интегрированную среду для удобной разработки приложений на основе Angular 2;
- создадим первый веб-компонент Angular 2 и поместим его в веб-страницу;
- добавим в веб-компонент простейшие интерактивные возможности;
- познакомимся с основными вспомогательными функциями форматирования выходных данных.

Обновленный подход

Как упоминалось выше, Angular 2 – это совершенно новый фреймворк Angular, основанный на архитектуре, изначально построенный на TypeScript, строгом надмножестве JavaScript, добавляющем статическую типизацию, поддержку интерфейсов и декораторов.

Другими словами, приложения, использующие Angular 2, основаны на архитектуре, включающей взаимосвязанные деревья веб-компонентов с их собственными интерфейсами ввода/вывода. Каждый компонент внутренне использует полностью переделанный механизм внедрения зависимостей. Справедливости ради следует отметить, что это лишь упрощенное описание того, чем Angular 2 яв-

ляется на самом деле. Однако даже простой проект, использующий Angular, наследует все его черты. В следующих главах мы познакомимся с организацией взаимодействий компонентов и внедрением зависимостей, после чего перейдем к маршрутизации, веб-формам и обмену информацией по протоколу HTTP. Это определение также объясняет, почему в книге не будет явных ссылок на Angular 1.x.

Очевидно, что нет смысла тратить время и место, ссылаясь на бесполезную для раскрытия описываемой темы информацию. Кроме того, поскольку предполагается, что читатель может быть незнаком с Angular 1.x, такие ссылки здесь явно излишни.

Веб-компоненты

Веб-компоненты представляют концепцию, которая охватывает четыре технологии, разработанные для создания элементов с высоким уровнем визуальной наглядности и возможностью повторного использования, что ведет к улучшению модульности, последовательности и облегчению обслуживания веб-приложений. К этим технологиям относятся:

- **шаблоны**: фрагменты разметки HTML, определяющие структуру отображаемого содержимого;
- **пользовательские элементы**: эти шаблоны содержат не только традиционные HTML-элементы, но и пользовательские оболочки, улучшающие визуальное представление элементов или расширяющие их функциональные возможности;
- **теневая модель DOM**: обеспечивает защищенную среду для инкапсуляции правил CSS-макета и JavaScript-модели поведения любого пользовательского элемента;
- **импорт HTML**: позволяет размещать в HTML-шаблоне не только HTML-элементы, но и другие HTML-документы.

Вообще говоря, компонент Angular 2 – это пользовательский элемент с шаблоном для размещения HTML-структуры макета, оформление которой определяется CSS-стилем, инкапсулированным в контейнер теневой модели DOM. Попробуем перевести это на обычный язык. Вспомним элемент управления HTML5, ограничивающий ввод числовых значений определенным диапазоном. Его удобно использовать, когда требуется дать возможность вводить значения, заключенные между двумя предопределенными границами. Если вы не использовали его раньше, вставьте следующий фрагмент разметки в пустой HTML-шаблон и загрузите его браузер:

```
<input id="mySlider" type="range" min="0" max="100" step="10">
```

Перед вами появится элемент управления в виде горизонтального ползунка. Если попробовать исследовать этот элемент с помощью инструментов разработчика в браузере, можно заметить скрытый набор HTML-тегов, отсутствующих в определении HTML-шаблона. Это и есть пример использования теневой модели DOM, когда действительный HTML-шаблон управляется встроенными в него CSS-правилами и дополнительными функциями управления ползунком. Согласитесь, что было бы достаточно сложно достичь того же результата самостоятельно. Самое замечательное, что Angular 2 предоставляет набор инструментов, необходимых для поддержки такой функциональности, благодаря чему мы можем строить собственные компоненты (элементы ввода, персонализированные теги и автономные виджеты), отображающие выбранную внутреннюю HTML-разметку с применением индивидуальных стилей, не влияющих (и не подверженных влиянию) на CSS-разметку страницы с такими компонентами.

Почему именно TypeScript?

Приложения на основе Angular 2 можно писать на разных языках, таких как: ECMAScript 5, Dart, ECMAScript 6, TypeScript или ECMAScript 7.

Язык TypeScript – это типизированное надмножество спецификации ECMAScript 6 (также известной как ECMAScript 2015), которое компилируется в обычный код на JavaScript и широко поддерживается современными ОС. Он придерживается объектно-ориентированного подхода, поддерживает аннотации, декораторы и контроль типов.

Причиной, почему в этой книге в качестве синтаксической основы выбран (и рекомендуется) язык TypeScript, является тот факт, что сам Angular 2 написан на этом языке. Опыт работы на TypeScript дает разработчику огромное преимущество при изучении внутреннего механизма фреймворка.

С другой стороны, поддержка аннотаций и контроля типов в TypeScript будет иметь первостепенное значение, когда мы дойдем до управления внедрением зависимостей и связями между компонентами.

Вообще говоря, проекты на основе Angular 2 можно реализовать, используя обычный синтаксис ECMAScript 6. Даже примеры, приведенные в книге, легко можно перенести на ES6, удалив аннотации типов и интерфейсов или заменив способ внедрения зависимостей в Typescript более сложными приемами, используемыми в ES6.



Все примеры в книге написаны исключительно на TypeScript, который мы рекомендуем использовать из-за выразительности аннотаций типов и более элегантного подхода к встраиванию зависимостей, основанного на контроле этих аннотаций типов.

Настройка рабочего окружения

Прежде чем начать реализацию первого компонента Angular 2, необходимо познакомиться со всеми инструментами, необходимыми для реализации программного обеспечения на TypeScript, не говоря уже о самих модулях фреймворка Angular 2.

Прежде всего создадим папку и проверим доступность в системе клиента NPM CLI, должным образом обновленного до последней стабильной версии. Если этот клиент у вас отсутствует, зайдите на сайт https://nodejs.org и установите последнюю версию Node.js.



На момент написания книги последней версией фреймворка была Angular 2 Release Candidate 1, поэтому представленные здесь примеры сборки и развертывания могут потерять актуальность. Автор поддерживает репозиторий по адресу https://github.com/deeleman/learning-angular2, из которого можно получить последнюю версию любого из примеров, содержащихся в этой книге. Репозиторий состоит из папок, соответствующих главам, и каждая папка содержит версии проекта для каждой главы. Если после установки или развертывания примеров возникнут проблемы, обратитесь к этому репозиторию.

Установка зависимостей

Очевидно, что подготовку рабочего окружения следует начать с установки самого фреймворка Angular 2 с его собственными зависимостями. Разработчики Angular 2 постарались сделать установку модульной, чтобы дать возможность включать в проекты только то, что действительно необходимо, в соответствии с предъявляемыми к нему требованиями.

Поэтому фреймворк Angular 2 поставляется не в виде единого установочного пакета, как это обычно бывает. Разработчику дается возможность выбрать только те модули, которые требуются для этого проекта, минимизировав общий объем зависимостей. Некоторые из пакетов, например `common` или `core`, обязательны независимо от вида разрабатываемого проекта. Другие пакеты, такие как `platform-browser-dynamic`, привязаны к виду проекта и целевой платформе.

Ниже приводится неполный список пакетов, которые чаще других используются в проектах:

- `@angular/core`: наиболее важный пакет, содержащий основу Angular и его наиболее общие элементы, такие как директивы и компоненты. Этот модуль всегда следует импортировать в проект;
- `@angular/common`: редко возникает необходимость явно импортировать элементы из этого модуля, но стоит отметить, что этот пакет содержит определения всех директив, служб и фильтров, встроенных в Angular 2, вместе с соответствующими классами;
- `@angular/compiler`: по аналогии с модулем `common` вам редко придется явно импортировать элементы из этого модуля. Он отвечает за компиляцию HTML-шаблонов в код, который позволяет отобразить пользовательский интерфейс приложения;
- `@angular/platform-browser`: содержит классы и функции, необходимые для компоновки и взаимодействия с моделью DOM в контексте веб-браузера. Этот модуль дает возможность обрабатывать такие обычные действия, как обновление заголовка страницы и настройка распознавания жестов. Этот пакет содержит также функции, необходимые для компиляции шаблонов в автономном режиме при производственной эксплуатации;
- `@angular/platform-browser-dynamic`: этот модуль часто будет упоминаться в книге, поскольку он предоставляет функцию загрузки, необходимую для инициализации приложений во время разработки;
- `@angular/http`: это HTTP-клиент Angular 2, который подробно рассматривается в *главе 6 «Асинхронная обработка данных в Angular 2»*;
- `@angular/router`: это встроенный в Angular 2 маршрутизатор, который на момент написания этой книги все еще находился в стадии бета-версии;
- `@angular/router-deprecated`: предыдущая версия встроенного маршрутизатора, обеспечивающая обратную совместимость с существующими приложениями. В *главе 7 «Маршрутизация в Angular 2»* мы подробно рассмотрим его основные отличия от нового маршрутизатора, еще находящегося на стадии разработки.

Ниже перечислены все сторонние библиотеки, необходимые как зависимости для проектов с Angular 2, помимо самих модулей Angular 2:

- `es6-shim`: включает полифиллы, обеспечивающие совместимость с ECMAScript 6 для старых движков JavaScript (главным образом для Microsoft Internet Explorer). Необходимость этой библиотеки объясняется отсутствием во многих популярных браузерах широкой поддержки ECMAScript 6, но, надеюсь, это скоро изменится. Некоторые другие реализации вместо этого модуля используют стандартную библиотеку `core-js`. В конце концов, можно выбрать любые полифиллы, обеспечивающие поддержку ядра программного интерфейса ES2015, требуемую Angular 2;
- `zone.js`: полифилл для спецификации Zone. Используется для обработки изменений, обнаруженных в приложениях на основе Angular 2;
- `reflect-metadata`: привносит поддержку декораторов в классах Angular 2 и метаданных компонентов. Использование декораторов будет рассмотрено ниже в этой же главе, а в *главе 2 «Введение в TypeScript»* будет представлен обзор различных их типов и реализаций. Декораторы являются основной частью Angular 2;
- `rxjs`: эта библиотека разработана в Microsoft Open Technologies, Inc. По утверждениям компании Microsoft, этот набор библиотек предназначен для создания асинхронных и основанных на событиях программ, использующих отслеживаемые коллекции и методы `Array#extras` из языка JavaScript. Короче говоря, библиотека RxJS служит для управления отслеживаемыми объектами и позволяет реализовать асинхронную обработку изменений в состоянии приложения. Работа с отслеживаемыми объектами в будущем станет стандартом, поддерживаемым всеми современными браузерами, поэтому со временем данная зависимость будет исключена.

Эти зависимости могут изменяться без предварительного на то уведомления, поэтому за актуальным списком требований обращайтесь к репозиторию GitHub.



Вас, вероятно, удивило количество библиотек, необходимых Angular 2, и тот факт, что эти зависимости не являются частью пакета Angular. Однако эти требования не являются специфическими для Angular 2, в настоящее время они характерны для подавляющего большинства современных приложений на JavaScript.

Для установки всех этих зависимостей и сторонних библиотек после создания папки проекта запустите в окне терминала следующие bash-команды:

```
$ mkdir learning-angular2
$ cd learning-angular2
$ npm init
$ npm install @angular/common @angular/core @angular/compiler --save
$ npm install @angular/platform-browser @angular/platform-browser-dynamic
--save
$ npm install @angular/router @angular/router-deprecated --save
$ npm install @angular/http --save
$ npm install es6-shim reflect-metadata rxjs zone.js --save
```

Помимо перечисленных зависимостей, для поддержки загрузки модулей и приведения их в соответствие с ES5 нужно также установить пакет универсального загрузчика модулей `systemjs`. Пакет `systemjs` не является единственным возможным вариантом управления загрузкой модулей в Angular 2. На самом деле его можно заменить другим загрузчиком, таким как **WebPack** (https://webpack.github.io/), но в этой книге все примеры используют SystemJS. Установим SystemJS, пометив его как зависимость для разработки, выполнив следующую команду:

```
$ npm install systemjs --save
```

Наконец, установим библиотеку **Bootstrap**, чтобы упростить создание привлекательного пользовательского интерфейса приложения, которое будет постепенно дорабатываться в каждой из глав книги. Это не является требованием Angular 2, а станет зависимостью проекта, реализуемого в этой книге:

```
$ npm install bootstrap --save
```

В процессе установки могут выводиться различные сообщения и предупреждения, что связано с версиями каждой из зависимостей, требуемых Angular 2 на данный момент времени, поэтому в случае появления вопросов я настоятельно рекомендую загрузить последнюю версию файла `package.json` из репозитория к этой книге: https://github.com/deeleman/learning-angular2/blob/master/chapter_01/package.json.

Загрузите этот файл в рабочий каталог и выполните команду `npm install`. NPM автоматически найдет и установит все зависимости.



Пользователям Mac OS, не обладающим правом на запись в каталог, где находится `npm` (обычно это каталог `/usr/local/bin/npm` или `/usr/local/npm` – для версий ОС, предшествующих Mac OS El Capitan), может потребоваться выполнить команду `npm install` с привилегиями суперпользователя.

Установка TypeScript

Итак, у нас имеется полный набор исходных текстов фреймворка Angular 2 и его зависимостей, а также пакеты Bootstrap, для придания привлекательного внешнего вида проекту, и SystemJS для управления загрузкой и связыванием модулей.

Однако в системе все еще отсутствует TypeScript. Установим TypeScript и сделаем его глобально доступным, чтобы получить удобный интерфейс командной строки для компиляции файлов:

```
$ npm install -g typescript
```

Отлично! Мы почти закончили. Осталось лишь уведомить TypeScript, что в проекте будет использоваться компилятор. Для этого просто выполните следующую команду:

```
$ tsc --init --experimentalDecorators --emitDecoratorMetadata --target ES5
--module system --moduleResolution node
```

По сути, эта команда просто добавляет в наш проект на TypeScript (который одновременно является и проектом Angular 2) поддержку экспериментальных декораторов (как уже упоминалось, это новая функция ES7 и TypeScript, широко используемая в Angular 2) и SystemJS как механизма по умолчанию для импорта модулей и зависимостей.

В результате этого действия в корне проекта появится новый файл `tsconfig.json`, включающий параметры, необходимые компилятору TypeScript для преобразования кода компонента в обычный код на JavaScript (соответствующий спецификации ECMAScript 5), понятный современным браузерам.



Не забывайте, что на настоящий момент браузеры не имеют встроенной поддержки ни TypeScript, ни ECMAScript 6, поэтому код должен быть преобразован в код на JavaScript, поддерживаемый целевыми браузерами.

Давайте взглянем на содержимое этого файла:

```
{
  "compilerOptions": {
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true,
    "target": "es5",
    "module": "system",
    "moduleResolution": "node",
    "noImplicitAny": false,
```

```
    "outDir": "built",
    "rootDir": ".",
    "sourceMap": false
  },
  "exclude": [
    "node_modules"
  ]
}
```

Все просто, не так ли? Свойства, включенные в манифест конфигурации, интуитивно понятны, но мне хотелось бы отметить три наиболее интересных из них:

- `rootDir`: указывает на папку, где компилятор будет искать TypeScript-файлы для компиляции (в данном случае это корневая папка примера);
- `outDir`: определяет, куда следует поместить скомпилированные файлы: если в командной строке компилятора отсутствует параметр `--outDir`, по умолчанию будет использоваться папка `built`, созданная во время выполнения там, где находится файл `tsconfig.json`;
- `sourceMap`: определяет режим сопоставления с исходным кодом для нужд отладки.

Присвойте этому свойству значение `true`, если нужно генерировать файлы сопоставления исходных кодов для обратной трассировки инструментами разработчика в браузере при появлении исключений.

Здесь также можно видеть, что папка `node_modules` добавлена в список исключений (свойство `exclude`), а это означает, что команда `tsc` будет пропускать эту папку и все ее содержимое при преобразовании TypeScript-файлов в ES5-файлы во всем дереве приложения.



Полное описание параметров компилятора TypeScript можно найти по адресу: https://github.com/Microsoft/TypeScript/wiki/Compiler-Option.

Установка утилиты typings

Помимо зависимостей проекта, таких как Bootstrap, и зависимостей Angular 2, пакету TypeScript требуются дополнительные библиотеки, облегчающие работу. В частности, ES6 добавляет в среду JavaScript новые методы и прикладные интерфейсы, которые должны быть описаны в компиляторе TypeScript. В противном случае он не распознает их как часть синтаксиса, что вызовет ошибки компиляции. Всякий раз, когда нужно оповестить компилятор TypeScript о программном

интерфейсе на JavaScript – встроенном или в виде сторонней библиотеки, используется файл определения типов TypeScript.

Обычно определения типов для TypeScript хранятся в файле с расширением `d.ts` (подробно об этом рассказывается в *главе 2 «Введение в TypeScript»*), который используется для проверки типов и предотвращения ошибок компиляции. Подготовка файлов с определениями типов в проектах не слишком сложна и требует лишь наличия специального инструмента: *typings*. Фактически мы обязаны установить файл с определениями типов, чтобы «подружить» компилятор TypeScript с последним прикладным интерфейсом ES6. Однако могу вас обрадовать: при установке инструмента управления определениями типов для TypeScript все необходимые данные можно получить прямо из реестра NPM и тем самым автоматизировать процесс поиска, установки и развертывания файлов с определениями типов. Для этого вернемся в консоль и выполним следующие команды:

```
$ npm install -g typings
$ typings install es6-shim --ambient --save
```

Первая команда устанавливает инструмент `typings` глобально, а вторая устанавливает в проект файл с определениями типов `es6-shim`, создавая при этом файл `typings.json`, который хранит ссылки на источники происхождения всех файлов, уже установленных и которые будут установлены позднее. Она создаст новую папку `typings` и поместит в нее требуемые файлы определений. Без них будут не доступны базовые возможности ES6, такие как функциональные методы класса `Array`. Прежде чем двинуться дальше, необходимо решить еще один вопрос, касающийся поддержки типов в TypeScript. При установке файлов с определениями типов генерируются два фасадных файла: `typings/main.d.ts` и `typings/browser.d.ts`. Однако компилятору TypeScript должен быть доступен только один из них. В противном случае дублирующие определения типов будут вызывать исключения. Поскольку разрабатывается интерфейсное приложение, отдадим предпочтение файлу `browser.d.ts` и отключим `main.d.ts`, а также связанные с ним определения файлов, поместив его в раздел `exclude` в файле `tsconfig.json`:

```
{
  "compilerOptions": {
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true,
    "target": "es5",
    "module": "system",
```

```
    "moduleResolution": "node",
    "noImplicitAny": false,
    "outDir": "built",
    "rootDir": ".",
    "sourceMap": false
  },
  "exclude": [
    "node_modules",
    "typings/main.d.ts",
    "typings/main"
  ]
}
```

Также рекомендуется исключить папку typings из дистрибутива проекта, включив ее в файл .gitignore, как это обычно делается с папкой node_modules. В дистрибутив требуется включить только манифест typings.json и выполнять установку файлов определений типов, как действие postinstall в файле package.json. Таким способом можно одной операцией установить и npm-зависимости, и файлы с определениями:

```
"scripts": {
  "typings": "typings",
  "postinstall": "typings install"
},
```

В этом случае пакет typings должен быть включен в package.json как зависимость для окружения разработки. Поэтому переустановим его с флагом --save-dev. И снова за последней версией файла package.json для этой главы обращайтесь к репозиторию GinHub.

Hello, Angular 2!

После установки пакета библиотек Angular 2 и полной поддержки TypeScript пришло время произвести тестирование. Во-первых, создадим и очистим файл hello-angular.ts (.ts – это расширение для TypeScript-файлов) в корневой папке проекта.



Здесь мы сталкиваемся с первым из множества соглашений, используемых в этой книге, касающимся именования файлов. Имена файлов модулей должны состоять из букв нижнего регистре с дефисом между словами. В *главе 5 «Создание приложений с помощью компонентов Angular 2»* мы подробно рассмотрим соглашения об именах и передовой опыт программирования при-

ложений с применением Angular 2. Но до тех пор будем придерживаться в учебных целях некоторых антишаблонов, что очень скоро заметят более опытные читатели.

Откроем файл и введем следующий текст:

```
import { Component } from '@angular/core';
import { bootstrap } from '@angular/platform-browser-dynamic';
```

Мы только что импортировали самые основные типы и функции, необходимые для создания очень простого компонента в следующем разделе. Синтаксис должен быть знаком тем, кто имел дело с ECMAScript 6. Если вы не принадлежите к их числу, не волнуйтесь, он будет подробно описан в *главе 2 «Введение в TypeScript»*.

Классы TypeScript

А теперь определим класс:

```
class HelloAngularComponent {
  greeting: string;
  constructor() {
     this.greeting = 'Hello Angular 2!';
  }
}
```

В спецификации ECMAScript 6 (и TypeScript) классы определяются как один из основных строительных блоков. Как видно из примера выше, класс содержит поле свойства `greeting` с типом `string`, заполняемое в конструкторе текстовой строкой. Функция-конструктор вызывается автоматически при создании экземпляра класса, и каждое свойство (и функция) должно быть аннотировано соответствующим ему типом (или типом значения, возвращаемого функцией).

Не стоит об этом сейчас беспокоиться. *Глава 2 «Введение в TypeScript»* содержит все необходимые пояснения, касающиеся механики TypeScript. Теперь же сосредоточимся на макете компонента. Как вы, вероятно, заметили, имя структуры соответствует другому общему соглашению, принятому в Angular 2. Имена классов определяются в стиле языка Pascal с добавлением суффикса, указывающего на его тип (то есть компонент, директива, фильтр и т. д.), в данном случае `Component`.

Введение в декораторы

Только что созданный класс контроллера уже позволяет создать экземпляр объекта со свойством `greeting`, но, чтобы превратить его в на-

стоящий компонент, к нему нужно добавить некоторый синтаксический сахар Angular 2. Ранее мы импортировали метаданные класса `Component`, помните? А теперь используем его, декорировав наш класс следующим образом:

```
@Component({
  selector: 'hello-angular',
  template: '<h1> {{greeting}} </h1>'
})
class HelloAngularComponent {
  greeting: string;
  constructor() {
      this.greeting = 'Hello Angular 2!';
  }
}
```

Декоратор – очень интересная экспериментальная особенность, появившаяся в спецификации ECMAScript 7, которая позднее была заимствована и реализована в TypeScript для добавления метаданных в классы. Существует несколько типов декораторов, и все они легко узнаваемы по префиксу @. *Глава 2 «Введение в TypeScript»* знакомит с декораторами, но подробное описание их базовой логики выходит за рамки этой книги. Тем не менее они будут использоваться на протяжении всей книги.

В данном случае мы сообщили компилятору, что класс `HelloAngularComponent` в действительности является компонентом Angular 2. Компонент предназначен для инкапсуляции пользовательского элемента `<hello-angular>`, свойство `template` которого определяет внутреннюю HTML-структуру компонента. Как уже упоминалось ранее, пользовательские элементы, инкапсулирующие HTML-шаблоны, являются основой веб-компонентов.

Компиляция TypeScript в понятный браузерам код на JavaScript

Фундаментом для нашего примера послужил TypeScript, но, к сожалению, вполне вероятно, что выбранный браузер не будет его поддерживать. Поэтому исходный код следует скомпилировать в старый добрый JavaScript ECMAScript 5.

Хочу вас обрадовать: интерфейс CLI TypeScript включает готовые инструменты для компиляции кода на TypeScript в JavaScript. Для этого откройте окно терминала, перейдите в каталог с файлом `hello-angular.ts` и введите следующую команду:

```
$ tsc --watch
```

Вновь созданный файл `hello-angular.js` с версией кода TypeScript, соответствующей спецификации ECMAScript 5, будет помещен в каталог `built` (или в каталог, указанный в свойстве `outDir` в файле `tsconfig.json`). Этот файл уже содержит некоторый функциональный код для поддержки настроенных метаданных декоратора.



Обратите внимание на наличие флага `--watch` в команде. Этот параметр информирует компилятор, что при внесении изменений в файл требуется автоматически выполнить повторную компиляцию. Опускайте этот флаг, когда нужна лишь одноразовая компиляция и нет необходимости отслеживать изменения в коде.

Компонент выглядит все лучше и лучше, и сейчас самый подходящий момент, чтобы начать его использовать, но прежде его нужно поместить куда-то, чтобы увидеть в действии. Давайте определим HTML-оболочку, или веб-контейнер, где он будет находиться.

HTML-контейнер

Создайте в корневом каталоге проекта HTML-файл с именем `index.html` и добавьте в него следующий код:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Hello Angular 2!</title>

    <script src="node_modules/es6-shim/es6-shim.min.js"></script>
    <script src="node_modules/zone.js/dist/zone.js"></script>
    <script src="node_modules/reflect-metadata/Reflect.js"></script>
    <script src="node_modules/systemjs/dist/system.js"></script>
    <script src="node_modules/rxjs/bundles/Rx.js"></script>

    <script src="systemjs.config.js"></script>
  </head>

  <body>

  </body>
</html>
```

Это простейшая версия HTML-контейнера для приложения, использующего Angular 2. Выглядит он достаточно просто, потому что

большая часть логики отображения и управление зависимостями обрабатываются самим компонентом.

Тем не менее обратите внимание на две вещи. Во-первых, в нем присутствует блок подключения сценариев. Он включает все зависимости полифилла ES2015 (ES6), реализации спецификаций Zone и Observables, обеспечивающих полную поддержку декораторов, и, наконец, функцию динамической загрузки модулей приложения.



Не пытайтесь изменить порядок строк в этих блоках кода, если не хотите столкнуться с неожиданными исключениями.

Во-вторых, сценарий включает ссылку на новый файл `systemjs.config.js`. Этот файл еще не написан, поэтому создайте его в корневой папке проекта для дальнейшей реализации. Рассмотрим его содержимое в следующих абзацах:

```
(function() {

  var pathMappings = {
    '@angular': 'node_modules/@angular',
    'rxjs': 'node_modules/rxjs',
  };

  var packages = [
    '@angular/common',
    '@angular/compiler',
    '@angular/core',
    '@angular/http',
    '@angular/platform-browser',
    '@angular/platform-browser-dynamic',
    '@angular/router',
    '@angular/router-deprecated',
    '@angular/testing',
    'rxjs',
    'built',
  ];

  var packagesConfig = {};

  packages.forEach(function(packageName) {
    packagesConfig[packageName] = {
      main: 'index.js',
      defaultExtension: 'js'
    };
  });
```

```
  System.config({
    map: pathMappings,
    packages: packagesConfig,
  });

})();
```

Как видите, файл `systemjs.config.js` содержит выражение с немедленно вызываемой функцией. То есть данный код будет выполнен в момент обработки его браузером. Рассмотрим поочередно каждую из частей этой процедуры:

```
var pathMappings = {
  '@angular': 'node_modules/@angular',
  'rxjs': 'node_modules/rxjs',
};
```

Здесь определяется объект со свойствами, содержащими полные пути. Далее, в этом же файле, данный объект будет использован для настройки псевдонимов путей в SystemJS, чтобы при выполнении оператора `import '@angular/core'` SystemJS импортировал основной пакет из `node_modules/@angular/core`. Однако пакеты не импортируются как единое целое. Необходимо иметь лишь точку входа в пакет, и, следовательно, нужно сообщить SystemJS, какой фасадный файл следует искать, чтобы импортировать весь модуль. Это достигается определением массива модулей с последующим созданием объекта, где пути к модулям являются ключами, содержащими точки входа, и объекта конфигурации с расширением файла по умолчанию для таких путей:

```
var packages = [
  '@angular/common',
  '@angular/compiler',
  '@angular/core',
  '@angular/http',
  '@angular/platform-browser',
  '@angular/platform-browser-dynamic',
  '@angular/router',
  '@angular/router-deprecated',
  '@angular/testing',
  'rxjs',
  'built',
];

var packagesConfig = {};
```

```
packages.forEach(function(packageName) {
  packagesConfig[packageName] = {
    main: 'index.js',
    defaultExtension: 'js'
  };
});
```

Полученная в результате переменная packagesConfig представляет собой объект с описанной выше конфигурацией. Она также включает главный файл для входа и расширение по умолчанию для папки сборки, куда компилятор TypeScript будет сохранять скомпилированные файлы.

В заключение необходимо настроить в SystemJS отображение путей, точку входа и расширения файлов по умолчанию, чтобы связать все пакеты с предоставленными ранее путями:

```
System.config({
  map: pathMappings,
  packages: packagesConfig,
});
```

Вот и вся настройка, необходимая SystemJS. Теперь можно начинать работу над приложением, опираясь на синтаксис ES2015 импорта модулей, как описывается в следующих разделах.

Сопроводительные примеры книги

Прежде чем заняться примером непосредственно, необходимо настроить локальный веб-сервер для его выполнения. Если у вас уже есть рабочий веб-сервер, который можно настроить на использование указанного выше каталога, тогда переходите к чтению следующего раздела. В противном случае вам следует подготовить такой веб-сервер для опробования примеров. Чтобы облегчить эту задачу, рекомендую установить чрезвычайно мощный и компактный модуль `lite-server` из NPM:

```
$ npm install -g lite-server
```

После этого вы сможете запускать веб-сервер, выполнив следующую команду в терминале из папки проекта:

```
$ lite-server
```

Эта команда запустит веб-сервер и откроет окно браузера со ссылкой на рабочий каталог. Все доступные параметры можно най-

ти в официальном репозитории модуля NPM (https://github.com/johnpapa/lite-server).



Пакет `lite-server` рекомендуется устанавливать вместе с пакетами `typescript` и `concurrently`, определив их как зависимости времени разработки с помощью флага `--save-dev`. Это даст возможность одновременно запускать компилятор TypeScript в режиме отслеживания и локальный сервер одной командой, которая может быть помещена в сценарий `start`, объявленный в файле `package.json`. После этого вы сможете приступить к сборке, перейдя в папку проекта и выполнив команду `npm start`. Такой подход реализован в примерах для этой книги, хранящихся в репозитории GitHub, так что можете смело загрузить оттуда удобный пример `package.json`.

Соединяем все вместе

Теперь HTML-файл готов для размещения компонента Angular 2. Для этого отредактируйте шаблон и поместите в него пользовательский элемент с тем же именем тега, которое было задано в свойстве `selector` компонента. Затем импортируйте файл с объявлением класса компонента, используя для этого программный интерфейс SystemJS. Сверьте внесенные изменения с кодом ниже:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Hello Angular 2!</title>
    <script src="node_modules/es6-shim/es6-shim.min.js"></script>
    <script src="node_modules/zone.js/dist/zone.js"></script>
    <script src="node_modules/reflect-metadata/Reflect.js"></script>
    <script src="node_modules/systemjs/dist/system.js"></script>
    <script src="node_modules/rxjs/bundles/Rx.js"></script>

    <script src="systemjs.config.js"></script>
      // Импортировать модуль компонента
      // без расширения файла
    <script>
      System.import ('built/hello-angular');
    </script>
  </head>
  <body>
    <!-- Пользовательский тег элемента -->
    <hello-angular></hello-angular>
  </body>
</html>
```

Круто, правда? Теперь мы можем создавать собственные элементы, которые делают все, что в них определено. Давайте откроем страницу в браузере, чтобы увидеть, как она действует, введя адрес http://localhost:3000/ (или указав другое имя хоста и номер порта, на которые настроен локальный веб-сервер).

К сожалению, если перезагрузить страницу в браузере, ничего не произойдет, и он будет отображать пустую страницу без содержимого. Чтобы что-то появилось, необходимо загрузить компонент и создать его экземпляр на HTML-странице.

Вернемся к файлу компонента `hello-angular.ts` и добавим в конец строку:

```
import { Component } from '@angular/core';
import { bootstrap } from '@angular/platform-browser-dynamic';

@Component({
  selector: 'hello-angular',
  template: '<h1> {{greeting}} </h1>'
})
class HelloAngularComponent {
  greeting: string;
  constructor() {
    this.greeting = 'Hello Angular 2!';
  }
}

bootstrap(HelloAngularComponent); // Компонент загружен!
```

Команда `bootstrap` создает экземпляр класса контроллера, передаваемого в аргументе, и включает его в приложение. В основном метод `bootstrap` выполняет следующие действия:

- анализирует конфигурацию компонента, переданного в первом аргументе, и проверяет его тип;
- находит в дереве DOM элемент с тегом, соответствующим свойству `selector` компонента;
- создает дочерний инструмент, который выполнит внедрение зависимостей в этот компонент и всех дочерних директив (включая компоненты, которые тоже являются директивами), что позволят использовать при размещении компонента подход, схожий с ветвлением дерева;
- создает новую зону. Зоны не будут рассматриваться в этой книге, просто отметим, что они отвечают за управление механизмом обнаружения изменений каждого загруженного экземпляра компонента в изолированном режиме;

- создает контекст теневой модели DOM в пользовательском элементе, идентифицируемом свойством `selector`, и отображает его с применением HTML-кода из шаблона компонента;
- сразу после создания экземпляра класса контроллера компонента вызывает механизм обнаружения изменений и затем инициализирует поставщиков, которые заполняют теневую модель DOM, внедряя необходимые данные.

Позже я расскажу, как с помощью команды `bootstrap` организовать отображение отладочной информации и переопределить поставщиков данных во всем приложении, чтобы инструмент внедрения зависимостей, встроенный в Angular 2, смог находить нужные зависимости.

Надеюсь, что вы запустили компилятор TypeScript в режиме отслеживания. Если это не так, выполните команду `tsc`, чтобы скомпилировать код, и перезагрузите страницу в браузере. А теперь можно порадоваться успешному выводу содержимого нашего первого компонента Angular 2 (рис. 1.1)!

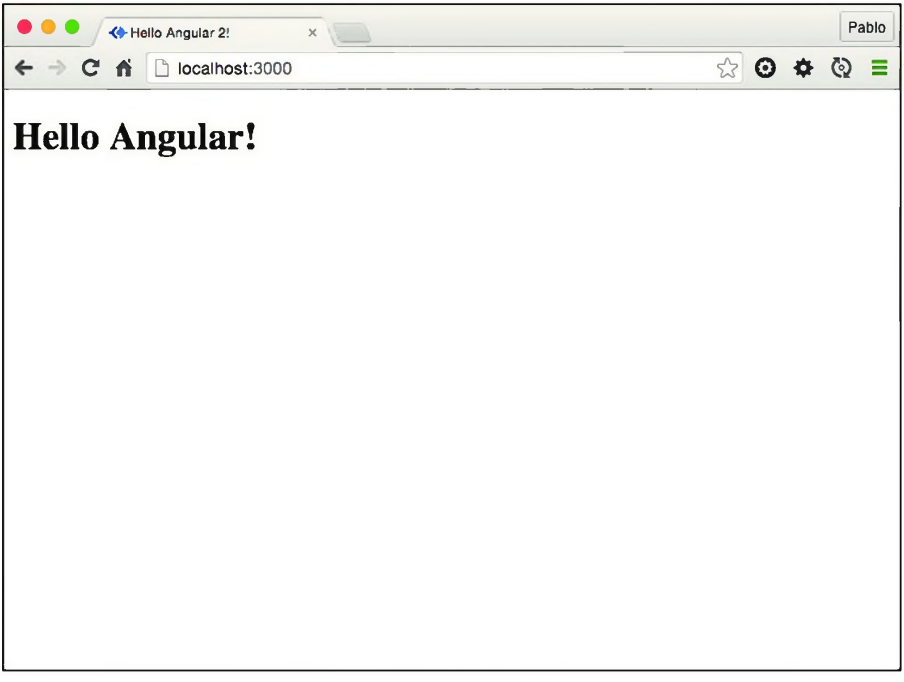


Рис. 1.1

Усовершенствование интегрированной среды разработки

Прежде чем продолжить путешествие по Angular 2, поговорим немного о среде разработки. Хороший редактор кода может стать бесценным союзником в процессе гибкой разработки, взяв на себя компиляцию кода TypeScript, статическую проверку типов и интроспекцию, а также предоставив функцию автозавершения кода и визуальные подсказки при отладке и сборке приложения. Поэтому кратко рассмотрим наиболее популярные редакторы кода и выберем из них тот, который лучше других сможет помочь нам при разработке приложений Angular 2. Если вы получаете удовольствие от запуска компилятора TypeScript в командной строке и не нуждаетесь в помощи при работе с кодом, можете смело переходить к чтению следующего раздела. Всем остальным я предлагаю окунуться в подразделы ниже, посвященные выбору подходящей интегрированной среды разработки.

Sublime Text 3

Это, пожалуй, один из самых распространенных редакторов кода в настоящее время, хотя в последнее время он несколько сбавил темп в расширении круга своих пользователей, которые все чаще отдают предпочтение другим быстро растущим конкурентам, таким как редактор Atom от GitHub. Если вы остановили выбор на этом редакторе, можно смело предположить, что он уже установлен в вашей системе вместе с Node (что очевидно, если только вы не установили TypeScript через NPM, как описывалось выше). Чтобы добавить в редактор поддержку TypeScript, нужно установить плагин TypeScript от Microsoft, доступный по адресу: https://github.com/Microsoft/TypeScript-Sublime-Plugin. Там же можно найти описание установки плагина, а также порядок настройки горячих клавиш.

После успешной установки плагина нажатие клавиш **Ctrl+пробел** приведет к отображению подсказки, основанной на интроспекции (рис. 1.2). Нажатие **F7** запустит процесс сборки и компиляции редактируемого файла в код на JavaScript. Еще одной замечательной особенностью, которую можно включить из меню, является оповещение об ошибках в коде, отображаемых в реальном времени.

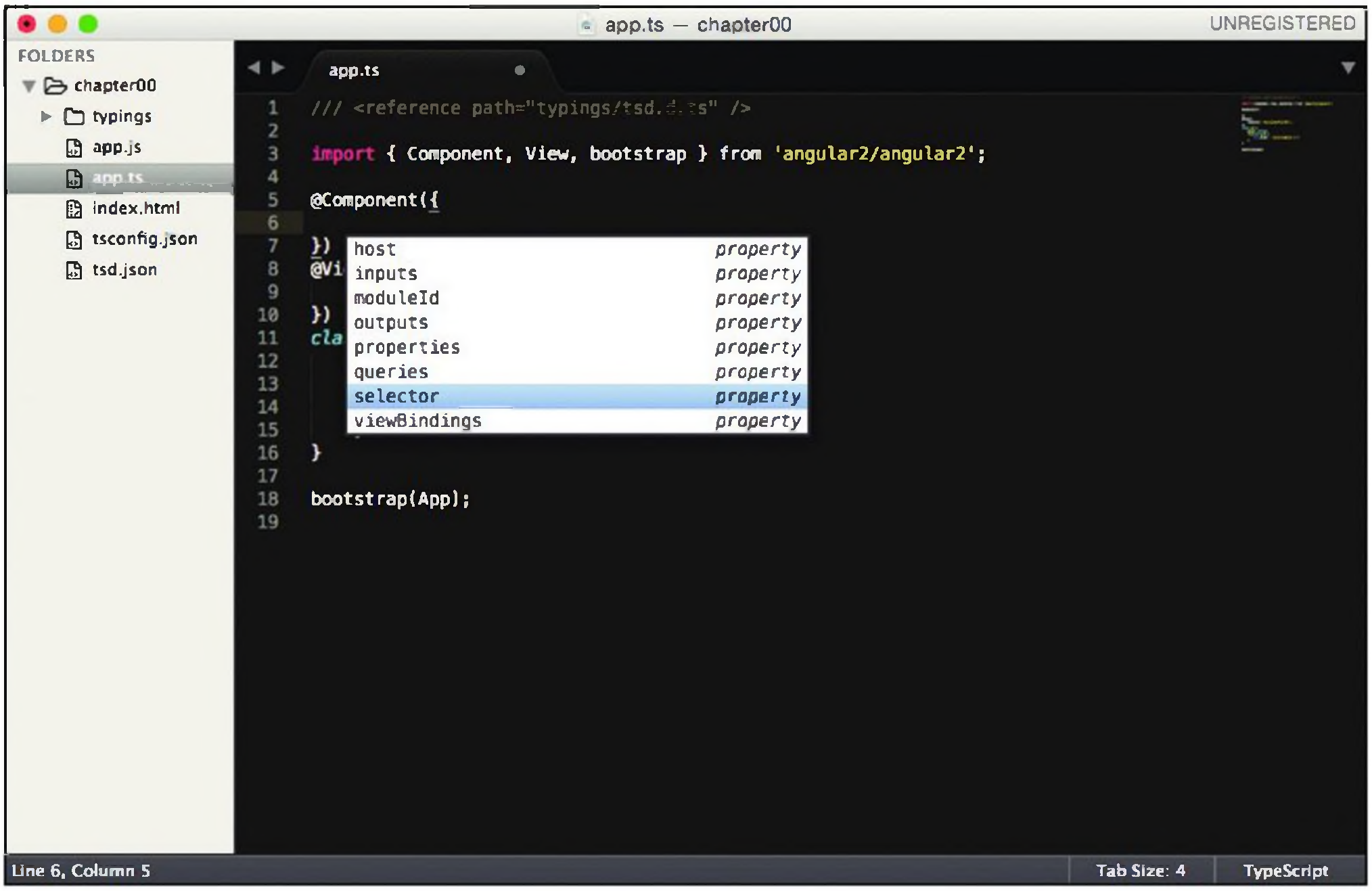


Рис. 1.2

Atom

Высокая гибкость и простота установки новых пакетов определили для многих пользователей выбор редактора Atom на роль интегрированной среды разработки. Кстати, примеры кода в этой книге были написаны только с помощью Atom.

Чтобы удобнее было работать с TypeScript при создании приложений на основе Angular 2, необходимо установить в редактор пакет TypeScript. Это можно сделать с помощью интерфейса APM CLI или просто использовать встроенный установщик пакетов. Доступные функции в основном совпадают с функциями в Sublime после установки пакета от Microsoft: автоматический вывод подсказок, статическая проверка типов, интроспекция кода и автоматическая сборка после сохранения – это лишь некоторые из них. Кроме того, пакет включает удобный встроенный генератор файлов `tsconfig.json`.

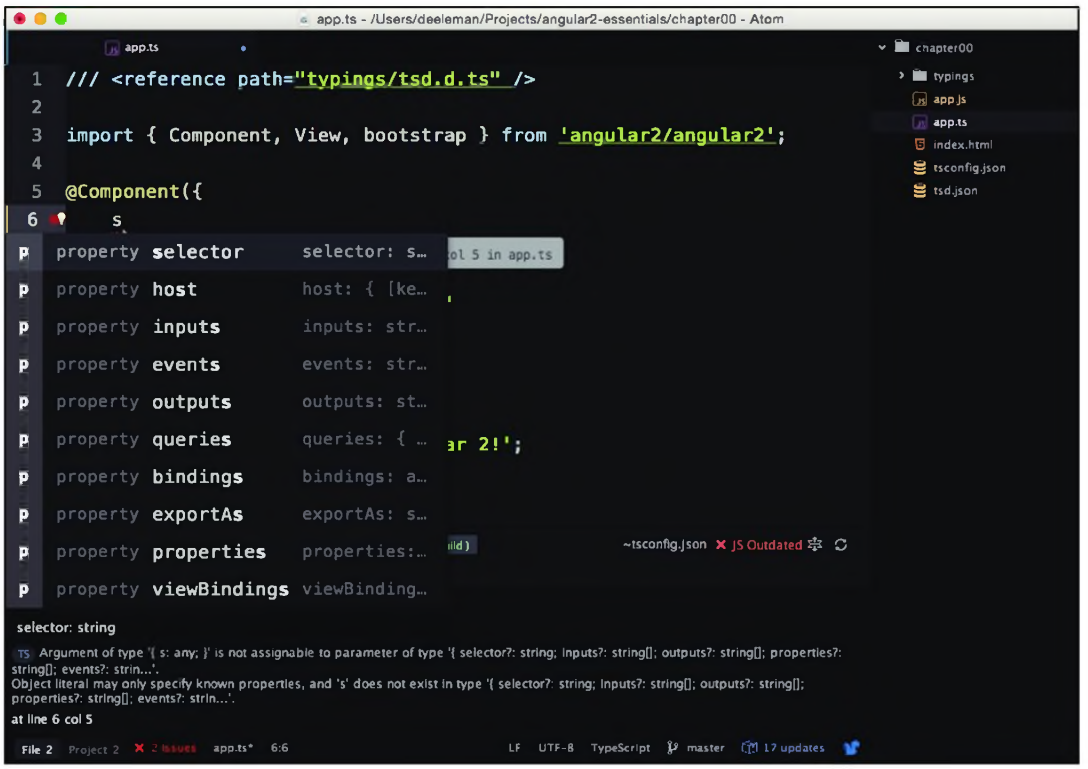


Рис. 1.3

Visual Studio Code

Visual Studio Code – относительно новый редактор кода, поддерживаемый Microsoft, набирающий силу и способный составить серьезную конкуренцию другим редакторам в мире Angular 2, главным образом благодаря отличной встроенной поддержке TypeScript. По сути, TypeScript является проектом корпорации Microsoft, поэтому вполне естественно, что один из популярных редакторов был задуман со встроенной поддержкой этого языка. Это означает, что все полезные свойства уже включены, в том числе подсветка синтаксиса, оповещение об ошибках и автоматическая сборка.

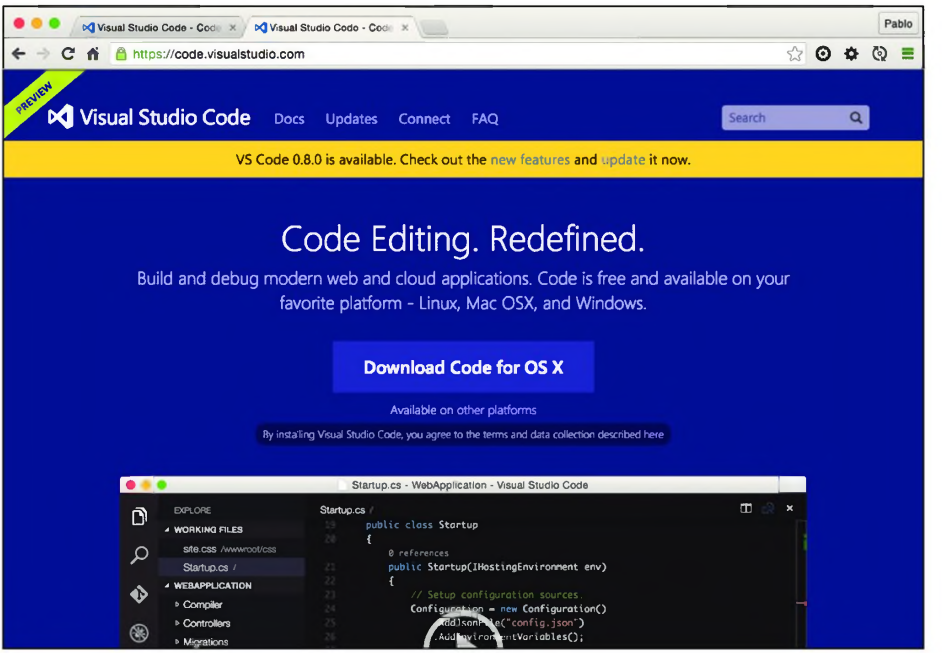


Рис. 1.4

WebStorm

Этот отличный редактор от IntelliJ также является хорошим выбором для разработчиков приложений на TypeScript. Интегрированная среда разработки поставляется со встроенной поддержкой TypeScript, поэтому в ней сразу можно приступать к разработке компонентов Angular 2. Кроме того, WebStorm имеет встроенный компилятор с поддержкой отслеживания файлов, что дает возможность компилировать код на TypeScript в обычный стандартный код на JavaScript, не полагаясь на дополнительные плагины от сторонних производителей.

Использование Gulp с другими интегрированными средами разработки

Возможно, ваша интегрированная среда разработки не была упомянута здесь и вы не желаете отказываться от своего редактора кода, не поддерживающего автоматическую компиляцию TypeScript в проектах. Или, может быть, вы не слишком комфортно себя чувствуете при работе с командами TypeScript в консоли.

Если это не так, смело переходите к чтению следующего раздела. Однако если это ваш случай, не волнуйтесь. Вам помогут современные средства выполнения задач на JavaScript. Давайте загрузим Gulp (http://gulpjs.com) и посмотрим, как создать очень простой сценарий автоматизации компиляции кода на TypeScript в проекте.

Во-первых, выполним установку зависимостей. По сути, достаточно установить Gulp и `gulp-typescript` – компилятор TypeScript для Gulp с поддержкой поэтапной компиляции. Введите следующие команды в окне консоли:

```
$ npm install -g gulp
$ npm install gulp gulp-typescript --save-dev
```

Создайте JavaScript-файл `gulpfile.js` в корневом каталоге проекта со следующим содержимым:

```
var gulp = require('gulp');
var ts = require('gulp-typescript');
var tsProject = ts.createProject('tsconfig.json');

gulp.task('build', function() {
  var tsResult = tsProject.src().pipe(ts(tsProject));
  return tsResult.js.pipe(gulp.dest('./built'));
});
```

В корневом каталоге проекта необходимо также создать файл `tsconfig.json`, откуда Gulp-задача будет извлекать настройки компиляции. После этого можно запустить обработку файлов, перечисленных в `tsconfig.json`, выполнив следующую команду:

```
$ gulp build
```

К сожалению, плагин `gulp-typescript` не поддерживает отслеживания файлов, поэтому для организации автоматической сборки при каждом изменении TypeScript-файла нужно воспользоваться встроенным методом `watch` из пакета Gulp. Для этого просто добавьте следующий фрагмент в конец файла `gulpfile.js`:

```
gulp.task('watch', ['build'], function() {
  gulp.watch('./**/*.ts', ['build']);
});
```

Теперь можно запустить процесс сборки и слежение за изменениями в файлах, выполнив следующую команду:

```
$ gulp watch
```

Погружение в компоненты Angular 2

Мы прошли долгий путь от ввода первого фрагмента кода на TypeScript до освоения базовой структуры компонентов Angular 2. Но, прежде чем переходить к более абстрактным вопросам, дополним текущий компонент новыми функциональными возможностями и выполним обзор наиболее характерных черт приложений и компонентов Angular.

Улучшение производительности

Иногда трудно бывает сконцентрироваться, особенно когда дело доходит до очень абстрактных вещей, требующих повышенного внимания. Широко известным способом решения этой проблемы является методика *Pomodoro technique* (техника помидора), следуя которой, подбирается список заданий, а затем задания разделяются на элементы, которые требуют для достижения нужного результата не больше 25 минут. Выбрав элемент задания, разработчик должен сосредоточиться на его решении, но не дольше чем на 25 минут, для чего используется таймер. Подробнее с этой методикой можно познакомиться на сайте: http://pomodorotechnique.com[1].

В этой книге мы создадим основной компонент с необходимой функциональностью и наполним его дополнительными функциями и элементами пользовательского интерфейса, обернутыми собственными компонентами. Для этого используем методику Pomodoro и начнем с создания таймера.

Методы компонента и обновление данных

Создайте новый файл `pomodoro-timer.ts` в той же папке и добавьте в него базовую реализацию очень простого компонента, как показано ниже. Пусть вас не беспокоит сложность кода, все изменения мы рассмотрим ниже:

```
import { Component } from '@angular/core';
import { bootstrap } from '@angular/platform-browser-dynamic';

@Component({
  selector: 'pomodoro-timer',
  template: '<h1> {{ minutes }}:{{ seconds }} </h1>'
})
```

---

[1] https://ru.wikipedia.org/wiki/Помидор_(метод). – *Прим. ред.*

```
class PomodoroTimerComponent {
  minutes: number;
  seconds: number;

  constructor() {
    this.minutes = 24;
    this.seconds = 59;
  }
}

bootstrap(PomodoroTimerComponent);
```

Справедливости ради стоит заметить, что новый компонент несильно отличается от созданного ранее. В классе `PomodoroTimerComponent` для большей ясности были изменены имена и определены два свойства, статически типизированные как числовые. В представлении они отображаются внутри элементов `<h1>`.

Теперь откроем файл `index.html` и заменим пользовательский элемент `<hello-angular></hello-angular>` новым тегом `<pomodoro-timer></pomodoro-timer>`. Файл `index.html` можно скопировать и сохранить под другим именем, если вы не хотите потерять разметку HTML с компонентом «Hello World».



Замечание об именовании пользовательских элементов

Селекторы в Angular 2 чувствительны к регистру символов. Как будет показано позже, компоненты являются подмножеством директив, способных поддерживать широкий спектр селекторов. При создании компонентов имена тегов в свойстве `selector` должны определяться в соответствии с соглашением, требующим разделять слова дефисом. Пользовательские теги в представлениях всегда должны завершаться закрывающим тегом, подобно непустым элементам. То есть `<custom-element></custom-element>` является правильной записью, а запись `<custom-element />` вызовет исключение. И последнее важное замечание: некоторые «типичные» имена в верблюжьей нотации могут конфликтовать с реализацией Angular 2, поэтому избегайте таких имен, как `AppView` или `AppElement`.

Обновите ссылку в блоке `System.import(...)` внутри файла `index.HTML`, чтобы она указывала на новый компонент:

```
System.import('built/pomodoro-timer')
      .then(null, console.error.bind(console));
```

Здесь следует отметить, что метод `import` из библиотеки SystemJS работает асинхронно и возвращает результат только после успешной

загрузки модуля. Этот возврат с задержкой можно использовать для вывода сообщения в консоль, что довольно удобно при отладке. Подробнее данная методика будет описана позже.

Если перейти в окно браузера и загрузить этот файл, в окне появятся числа, объявленные в классе компонента. Но нам требуется нечто большее, чем простое отображение нескольких чисел, не так ли? В действительности нам требуется реализовать обратный отсчет времени, и этого можно достичь, внеся следующие изменения. Сначала определим функцию обновления счетчика, которая будет вызываться в цикле. Добавим ее после функции конструктора:

```
tick(): void {
  if (--this.seconds < 0) {
    this.seconds = 59;
    if (--this.minutes < 0) {
      this.minutes = 24;
      this.seconds = 59;
    }
  }
}
```

Как видите, функции в TypeScript должны аннотироваться типом возвращаемого значения или типом `void`, если ничего не возвращают. Функция выше оценивает текущие значения минут (`minutes`) и секунд (`seconds`) и затем либо уменьшает их, либо сбрасывает в начальное состояние. Эта функция вызывается каждую секунду с помощью интервального таймера, запускаемого в конструкторе класса:

```
constructor() {
  this.minutes = 24;
  this.seconds = 59;
  setInterval(() => this.tick(), 1000);
}
```

Здесь первый раз мы использовали стрелочную функцию (такие функции также называют лямбда-функциями, жирными стрелками и т. д.). Этот новый синтаксис функций, введенный в спецификации ECMAScript 6, более подробно будет рассматриваться в *главе 2 «Введение в TypeScript»*. Кроме того, функция `tick` помечена как `private`, поэтому она недоступна за пределами экземпляра объекта `PomodoroTimerComponent`.

Пока все в порядке! Таймер Pomodoro выполняет обратный отчет от 25 минут до 0, а затем все начинается заново. Проблема заключа-

ется в том, что в двух местах мы имеем повторяющийся код. Давайте перепишем пример, чтобы избежать дублирования.

```
constructor() {
  this.resetPomodoro();
  setInterval(() => this.tick(), 1000);
}

resetPomodoro(): void {
  this.minutes = 24;
  this.seconds = 59;
}

private tick(): void {
  if (--this.seconds < 0) {
    this.seconds = 59;
    if (--this.minutes < 0) {
      this.resetPomodoro();
    }
  }
}
```

Мы заключили инициализацию (и сброс) минут и секунд в функцию `resetPomodoro`, которая вызывается после создания экземпляра компонента и по достижении конца отсчета. Хотя подождите! Согласно методике Pomodoro, практикующимся разрешено отдыхать между сеансами Pomodoro или даже ставить их на паузу, если вдруг возникнут какие-то неожиданные обстоятельства. То есть мы должны обеспечить некоторую интерактивность, чтобы пользователь мог запускать, приостанавливать и возобновлять текущий Pomodoro-таймер.

Добавление интерактивности в компонент

Angular 2 обеспечивает отличную поддержку событий через декларативный интерфейс, напоминающий подобные интерфейсы в других фреймворках, таких как React. Сначала изменим определение шаблона:

```
@Component({
  selector: 'pomodoro-timer',
  template: `
    <h1> {{ minutes }}:{{ seconds }} </h1>
    <p>
      <button (click)="togglePause()">
```

```
        {{ buttonLabel }}
      </button>
    </p>
  `
})
```

Здесь использован прием определения многострочного текста! Спецификация ECMAScript 6 ввела понятие *шаблонных строк* – строковых литералов с поддержкой встроенных выражений и многострочным содержимым. Более подробно они рассматриваются в *главе 2 «Введение в TypeScript»*.

Сейчас же просто констатируем факт добавления нового фрагмента HTML с кнопкой и обработчиком события щелчка на кнопке, вызывающим метод `togglePause`. Атрибут `click` может быть вам незнаком, несмотря на то что он полностью соответствует стандартам W3C. Более подробно об этом рассказывается в *главе 3 «Реализация свойств и событий компонентов»*. Рассмотрим метод `togglePause` и новую метку `buttonLabel`, но прежде изменим свойства класса так, чтобы они выглядели следующим образом:

```
class PomodoroTimerComponent {
  minutes: number;
  seconds: number;
  isPaused: boolean;
  buttonLabel: string;

  // ...Остальной код не изменился
}
```

Мы добавили два новых поля. Первое поле – `buttonLabel` – содержит текст, отображаемый на кнопке. Второе поле – `isPaused` – будет содержать значение `true/false` в зависимости от состояния таймера. Соответственно, значения этих полей должны где-то изменяться. Для этого создадим метод `togglePause`, упоминавшийся выше:

```
togglePause(): void {
  this.isPaused = !this.isPaused;
  // если начат обратный отсчет
  if (this.minutes < 24 || this.seconds < 59) {
    this.buttonLabel = this.isPaused ? 'Resume' : 'Pause';
  }
}
```

Метод `togglePause` просто изменяет значение переменной `isPaused` на противоположное, а затем в зависимости от нового значения и со-

стояния таймера (если он был запущен, значение времени в переменных `minutes` и `seconds` будет меньше начального значения) устанавливается нужная надпись для кнопки.

А теперь нужно инициализировать эти значения, выбрав для этого самое удобное место – функцию `resetPomodoro`:

```
resetPomodoro(): void {
  this.minutes = 24;
  this.seconds = 59;
  this.buttonLabel = 'Start';
  this.togglePause();
}
```

Вызов метода `resetPomodoro` сбросит компонент в исходное состояние, благодаря чему всякий раз, когда компонент достигнет состояния, требующего сброса, обратный отсчет будет остановлен. Осталось добавить только один хитрый прием в метод контроллера, обрабатывающего обратный отсчет:

```
private tick(): void {
  if (!this.isPaused) {
    this.buttonLabel = 'Pause';
    if (--this.seconds < 0) {
      this.seconds = 59;
      if (--this.minutes < 0) {
        this.resetPomodoro();
      }
    }
  }
}
```

Очевидно, что когда таймер находится в режиме паузы, обратный отчет должен быть приостановлен, поэтому мы завернули реализацию метода в условие. Кроме того, когда таймер активен, на кнопке должен отображаться другой текст, а когда отсчет достигнет своего конца, таймер следует остановить и сбросить компонент в исходное состояние, чтобы получить ожидаемое поведение. Именно для этого в метод `resetPomodoro` добавлен вызов функции `togglePause`.

Улучшения для вывода данных в представлении и доработка пользовательского интерфейса

А теперь перезагрузите страницу в браузере и поэкспериментируйте с функцией приостановки/возобновления отсчета. Надеюсь, вы за-

метили необходимость некоторой доработки: когда счетчик секунд достигает значений меньше 10, отображаются однозначные числа вместо двузначных, которые мы привыкли видеть в цифровых и наручных часах. К счастью, Angular 2 предоставляет множество декларативных помощников, форматирующих вывод данных в шаблонах. Они называются *фильтрами* (pipes) и более подробно рассматриваются в *главе 3 «Реализация свойств и событий компонентов»*. Сейчас мы просто добавим фильтр `number` в шаблон компонента и настроим формат вывода секунд из поля `seconds` в виде двух цифр. Измените шаблон, как показано ниже:

```
@Component({
  selector: 'pomodoro-timer',
    template: `
      <h1> {{ minutes }}:{{ seconds | number: '2.0' }} </h1>

      <p>
        <button (click)="togglePause()">
          {{ buttonLabel }}
        </button>
      </p>
    `
})
```

Здесь к имени переменной мы добавили имя фильтра, отделив его символом вертикальной черты (|). Загрузив шаблон, можно убедиться, что секунды теперь всегда отображаются в виде двузначного числа.

Итак, мы закончили создание полноценного виджета Pomodoro-таймера, который можно встраивать в более сложные приложения. В *главе 5 «Создание приложений с помощью компонентов Angular 2»* будет описан процесс встраивания и вложения компонентов в контекст крупных деревьев компонентов.

Тем не менее давайте немного доработаем пользовательский интерфейс, сделав компонент более привлекательным. Ранее мы уже добавили атрибут `class` в тег кнопки, предвосхитив включение в проект CSS-фреймворка Bootstrap. Импортируем стили, загруженные с по-

мощью NPM во время установки зависимостей проекта. Откройте файл pomodoro-timer.html и добавьте следующий код в конец элемента <HEAD>:

```
<link rel="stylesheet" href="node_modules/bootstrap/dist/css/bootstrap.min.css">
```

А теперь приукрасим пользовательский интерфейс, добавив красивый заголовок прямо перед компонентом:

```
<body>
  <nav class="navbar navbar-default navbar-static-top">
    <div class="container">
      <div class="navbar-header">
        <strong class="navbar-brand">My Pomodoro Timer</strong>
      </div>
    </div>
  </nav>
  <pomodoro-timer></pomodoro-timer>
</body>
```

Настроим кнопку компонента с помощью класса из фреймворка Bootstrap, придав ему большую индивидуальность, и обернем весь шаблон в центрирующий контейнер с красивым значком сверху. Для этого внесите следующие изменения в шаблон:

```
<div class="text-center">
  <img src="assets/img/pomodoro.png" alt="Pomodoro">
  <h1> {{ minutes }}:{{ seconds | number: '2.0' }} </h1>
  <p>
    <button (click)="togglePause()" class="btn btn-danger">
      {{ buttonLabel }}
    </button>
  </p>
</div>
```

На роль значка было выбрано изображение помидора. Вы можете использовать любое другое изображение по своему вкусу или вообще убрать значок, хотя в следующих главах он понадобится. На рис. 1.5 представлен внешний вид приложения таймера Pomodoro после реализации всех этих визуальных настроек:

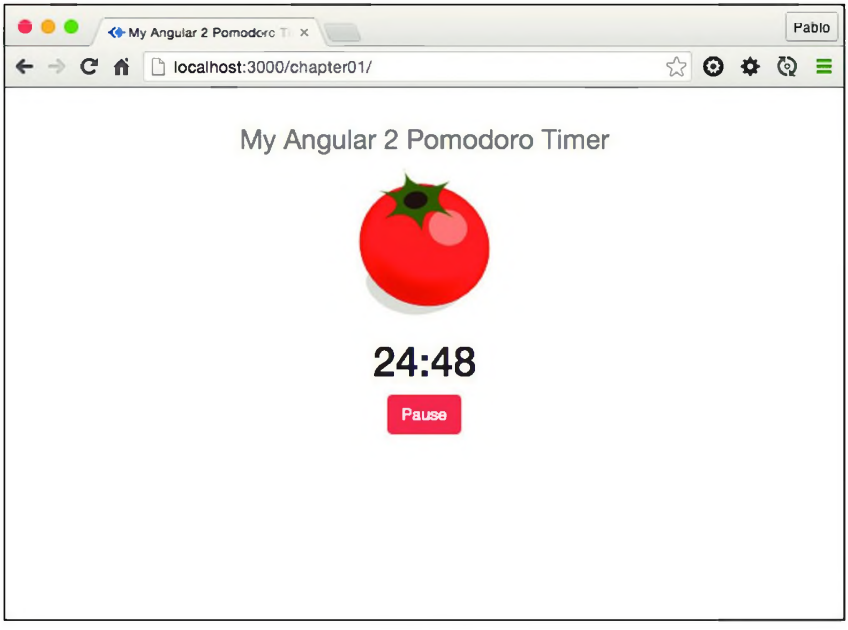


Рис. 1.5



Загрузка исходного кода примеров

Получить файлы с примерами кода можно по адресу: https://github.com/deeleman/learning-angular2.

Также файлы примеров кода для этой книги можно получить на сайте: http://www.packtpub.com. Где бы вы не купили эту книгу, вы можете посетить страницу http://www.packtpub.com/support и зарегистрироваться для получения файлов по электронной почте.

Для скачивания файлов кода необходимо выполнить следующие действия:

- войдите или зарегистрируйтесь на веб-сайте, указав свой адрес электронной почты и пароль;
- наведите указатель мыши на вкладку **SUPPORT** вверху;
- щелкните на ссылке **Code Downloads & Errata**;
- введите название книги в поле **Search**;
- выберите книгу, для которой вы хотите скачать файлы с примерами;
- в раскрывающемся меню укажите, где вы приобрели эту книгу;
- щелкните на ссылке **Code Download**, чтобы скачать примеры.

Скачать файлы можно также, щелкнув на кнопке **Code Files** на веб-странице книги сайта издательства Packt. Чтобы перейти на эту

страницу, введите название книги в поле **Search**. Обратите внимание, что перед этим требуется войти в свою учетную запись Packt. После загрузки архива извлеките из него файлы с помощью последней версии программы:

- WinRAR / 7-Zip для Windows;
- Zipeg / iZip / UnRarX для Mac;
- 7-Zip / PeaZip для Linux.

Итоги

Мы познакомились с веб-компонентами, которые определяются современными веб-стандартами, и простым программным интерфейсом Angular 2, дающим нам возможность создавать собственные компоненты. Кроме того, в качестве подготовки к *главе 2 «Введение в TypeScript»* мы познакомились с некоторыми особенностями языка TypeScript. Мы увидели, как настроить рабочую среду, включающую компилятор TypeScript и библиотеки Angular 2 для использования в проектах, а также исследовали роль каждой зависимости в приложении.

Первый компонент дал возможность обсудить форму класса контроллера со свойствами, конструктором и вспомогательными функциями и показать важность аннотаций с метаданными в контексте приложений Angular 2 для интеграции компонента в HTML-среду. Теперь мы знаем, как развернуть веб-сервер и как добавить в редактор поддержку синтаксиса TypeScript и контроль типов, чтобы упростить разработку приложений Angular 2. Наш первый веб-компонент имеет собственный шаблон, декларативно описывающий связи со свойствами в форме выражений интерполяции переменных и осуществляющий форматирование с применением фильтра. Подключение обработчиков событий теперь реализуется проще, чем когда-либо, для чего используется стандартный синтаксис.

Следующая глава подробно описывает все возможности языка TypeScript, знать которые необходимо для успешной работы с Angular 2.

Глава 2

Введение в TypeScript

В предыдущей главе мы создали первый компонент, использовав язык TypeScript для реализации сценария, придающего компоненту нужную форму. Все примеры в книге используют только такой синтаксис. Как можно будет убедиться в дальнейшем, написание сценариев на TypeScript и использование статической типизации обеспечивают значительное преимущество относительно других языков.

Эта глава не является полным обзором TypeScript. Здесь внимание будет уделено лишь основным элементам языка, и их рассмотрение будет привязано к работе с Angular 2. Дополнительно хочу отметить, что язык TypeScript довольно прост, поэтому мы охватим большую часть его основных элементов.

В этой главе рассматриваются следующие темы:

- происхождение и логическое обоснование TypeScript;
- знакомство с интернет-ресурсами, позволяющими закрепить практические навыки;
- краткое описание идеи типизированных значений и их представления;
- создание собственных типов, основанных на классах и интерфейсах;
- организация архитектуры приложения с помощью модулей.

Причины создания TypeScript

Естественная эволюция ранних небольших веб-приложений на JavaScript в огромные монолитные клиенты выявила недостатки спецификации ECMAScript 5 языка JavaScript. В крупномасштабных приложениях на JavaScript, по мере увеличения их размера и сложности, возникают серьезные проблемы, связанные с надежностью и масштабируемостью.

Эта проблема становится еще более актуальной при необходимости интеграции в приложение новых библиотек и модулей. Отсутствие хороших механизмов взаимодействий привело к очень громоздким решениям, которые перестали соответствовать предъявляемым требованиям.

В ответ появилась спецификация **ECMAScript 6** (другие названия **ES6** и **ES2015**), обещающая решить вопросы надежности и масштабируемости путем введения улучшенной загрузки модулей и архитектуры языка для большей управляемости и широкого спектра удобных синтаксических конструкций для облегчения управления типами и объектами. Введение основанного на классах программирования позволяет применить объектно-ориентированный подход при разработке крупных приложений.

Программисты Microsoft потратили почти 2 года на реализацию надмножества языка, сочетающего соглашения из ES6 с некоторыми возможностями, заимствованными из ES7. Идея заключалась в том, чтобы создать нечто, что упростит разработку корпоративных приложений за счет снижения вероятности появления ошибок с помощью статического контроля типов, улучшенной оснастки и анализа кода.

После 2 лет разработки, возглавляемой Андерсом Хейлсбергом (Anders Hejlsberg), ведущим архитектором C#, создателем Delphi и Turbo Pascal, в 2012 году появился TypeScript 0.8, который достиг версии 1.0 только два года спустя. TypeScript не только опередил ECMAScript 6, но и реализовал ее возможности, предоставив надежную среду для построения крупномасштабных приложений путем введения, кроме всего прочего, необязательного статического определения типов через аннотации, тем самым обеспечив контроль типов во время компиляции. Это способствует выявлению ошибок на ранних стадиях разработки. Поддержка файлов деклараций дает разработчикам возможность описания интерфейсов их модулей, чтобы другие могли лучше интегрировать их в свой код.

Преимущества TypeScript

Приведенная на рис. 2.1 инфограмма дает общее представление о возможностях, отличающих спецификацию ECMAScript 6 от ECMAScript 5, а также об их отличии от языка TypeScript.

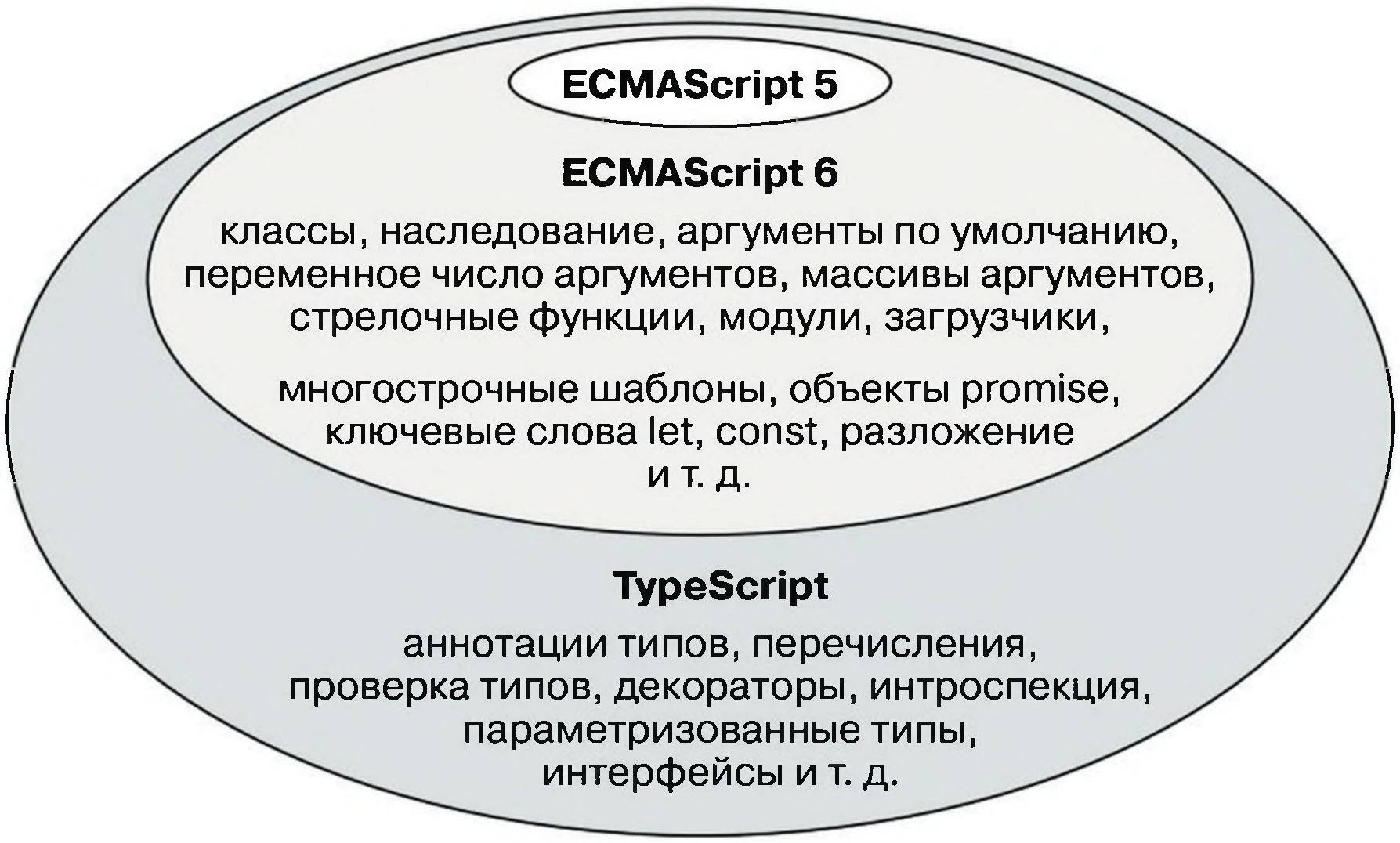


Рис. 2.1

Одним из главных преимуществ языка TypeScript как надмножества спецификации ECMAScript 6 в вашем следующем проекте станет низкий входной ценз. Если вы владеете ECMAScript 6, можете ограничиться этим, поскольку все дополнительные возможности TypeScript являются необязательными. Вы можете выбирать и использовать на практике только возможности, помогающие вам достичь конкретной цели. В итоге существует длинный список весомых аргументов в пользу применения TypeScript в вашем следующем проекте, и все они, очевидно, применимы к Angular 2. Вот лишь некоторые из них:

- применение аннотаций типов обеспечивает последовательную интеграцию различных модулей, улучшает читаемость кода и его наглядность;
- встроенная проверка типов позволяет анализировать код во время компиляции и выявлять ошибки до его выполнения;
- использование типов обеспечивает согласованность внутри приложений. Сочетание двух предыдущих аргументов ведет в долгосрочной перспективе к уменьшению количества ошибок в коде;
- TypeScript расширяет классы долгожданными возможностями, такими как поля класса, закрытые свойства и методы, перечисления и т. д.;
- декораторы открывают возможность расширения классов и их реализации беспрецедентными способами;

- создание интерфейсов и файлов с определениями типов (которые не рассматриваются в книге) обеспечивает более гладкую и бесшовную интеграцию библиотек в другие системы и приложения;
- различные интегрированные среды разработки имеют потрясающую поддержку TypeScript с подсветкой синтаксиса, проверкой типов в реальном времени и автоматической компиляцией;
- синтаксис TypeScript, безусловно, поможет разработчикам, переходящим с других языков, таких как Java, C#, C++ и др.

Знакомство с ресурсами, посвященными TypeScript

В предыдущей главе рассказывалось, как установить TypeScript в системе и использовать компилятор для преобразования TypeScript-файлов в файлы сценариев, совместимые со спецификацией ES5. Теперь мы узнаем, где можно получить дальнейшую поддержку и проверить знание языка TypeScript.

Официальный сайт TypeScript

В первую очередь, естественно, следует упомянуть официальный сайт языка: http://www.typescriptlang.org[1]. Там можно найти более обширное введение в язык, ссылки на интегрированные среды разработки и сайты корпоративных сторонников этого проекта. Однако наиболее важными разделами, к которым мы часто будем обращаться, являются раздел обучения и раздел с песочницей для экспериментов.

Раздел **Learn it fast** (Краткое знакомство) содержит обучающие материалы, позволяющие ускорить освоение языка. Его можно использовать, чтобы вспомнить, о чем шла речь в предыдущей главе, но мы рекомендуем пропустить его, перейдя на страницы с примерами и описанием языка, причем последняя является прямой ссылкой на полную обширную документацию в репозитории GitHub. Это бесценный ресурс как для новичков, так и для опытных пользователей.

Раздел **Playground** (Песочница) предлагает удобную возможность поэкспериментировать и опробовать некоторые готовые примеры кода, охватывающие наиболее распространенные особенности языка. Вы можете использовать этот инструмент для тестирования примеров кода из этой главы.

---

[1] Перевод на русский язык руководства по языку TypeScript можно найти по адресу: http://typescriptrus.blogspot.ru/. – *Прим. ред.*



Рис. 2.2

Сайт TypeScript Wiki

В предыдущей главе уже была дана ссылка на сайт TypeScript Wiki, когда речь шла о базовых параметрах, которые необходимо знать при работе с командами компилятора TypeScript.

Исходный код TypeScript открыт для общего доступа в репозитории GitHub, и команда Microsoft затратила немало усилий на документирование различных его аспектов, разместив эту документацию в Wiki-формате на сайте репозитория. Обращайтесь туда при появлении вопросов и желании подробно ознакомиться с любой из возможностей языка или формальным аспектом его синтаксиса.

Wiki-хранилище можно найти на странице: https://github.com/Microsoft/TypeScript/wiki.

Типы в TypeScript

Работа с TypeScript или с любым другим языком программирования в основном подразумевает работу с данными, а такими данны-

ми может быть самая разная информация. Под типами подразумеваются существительные, отражающие тот факт, что данные могут иметь вид текстовой строки, целочисленного значения или массива значений этих типов. В этом нет ничего нового для JavaScript, так как он всегда поддерживал неявную работу с типами, только более гибким образом. Это означает, что любой конкретной переменной можно присвоить (или вернуть из функции) значение любого типа. Иногда это приводило к ошибкам и исключениям, поскольку происходил конфликт типов между тем, что возвращено, и тем, что ожидалось. Такая гибкость все еще может обеспечиваться с помощью типа `any`, как будет показано ниже в этой главе, тем не менее статическая типизация переменных позволяет разработчикам и их интегрированным средам разработки иметь полную картину о том, какие данные предполагается обрабатывать в каждом фрагменте кода. Этот подход бесценен своей помощью в отладке приложения еще во время компиляции, до того, как станет слишком поздно.

Тип string

Пожалуй, одним из самых широко используемых простых типов является тип `string`, применяемый, когда переменной присваивается фрагмент текста:

```
var brand: string = 'Chevrolet';
```

Обратите внимание, что определение типа следует за именем переменной и отделяется от него символом двоеточия. Такая аннотация типов в TypeScript уже использовалась в предыдущей главе.

Определяя фактические значения типа `string`, можно использовать как одинарные, так и двойные кавычки, что соответствует спецификации ECMAScript6. Имеется также возможность определять многострочные текстовые значения с поддержкой интерполяции:

```
var brand: string = 'Chevrolet';
var message: string = `Today it's a happy day!
  I just bought a new ${brand} car`;
```

Объявление переменных в ECMAScript 6

TypeScript как надмножество ECMAScript 6 поддерживает наглядное объявление сущностей с помощью ключевого слова `let`, информирующее о том, что область существования переменной ограничена вмещающим ее объявление блоком (либо телом цикла `for`, либо

любым другим блочным оператором). С другой стороны, ключевое слово const указывает, что переменные, объявленные таким способом, всегда имеют один и тот же тип и значение. В остальной части этой главы для объявления переменных будет применяться традиционная нотация var, но и о прочих формах не стоит забывать.

Тип number

Тип number является еще одним распространенным простым типом данных наряду с типами string и boolean. Так же как в JavaScript, тип number определяет число с плавающей запятой. Шестнадцатеричные, десятичные, двоичные и восьмеричные литералы также имеют тип number:

```
var age: number = 7;
var height: number = 5.6;
```

Тип boolean

Тип boolean определяет данные, которые могут иметь значение true или false, представляющее выполнение некоторого условия:

```
var isZeroGreaterThanOne: boolean = false;
```

Массивы

Обработки исключений, возникающих из-за присваивания элементам массивов значений несовместимых типов, теперь легко избежать, определив тип массива, который указывает, что массив может содержать только элементы определенного типа. Синтаксис требует указывать постфикс [] в аннотации типа:

```
var brands: string[] = ['Chevrolet', 'Ford', 'General Motors'];
var childrenAges: number[] = [8, 5, 12, 3, 1];
```

Если попытаться добавить в массив childrenAges новый элемент с типом, отличным от number, компилятор выведет сообщение об ошибке. Такое поведение гарантирует согласованность типов элементов и отсутствие ошибок в коде.

Динамическая типизация с помощью типа any

Иногда трудно определить тип данных, основываясь на имеющейся информации, особенно когда в TypeScript применяется устаревший код или выполняется интеграция со слабо типизированными библио-

теками и модулями от сторонних производителей. Но это не повод для волнений, TypeScript предлагает удобный тип и для этих случаев. Тип `any` совместим со всеми другими типами, поэтому переменные этого типа можно инициализировать любыми значениями и в дальнейшем присваивать им любые значения. Но вместе с большими возможностями приходит и большая ответственность. Отказ от преимуществ статического контроля типов влечет увеличение вероятности появления ошибок, связанных с типами, при прохождении данных через модули, и разработчик должен сам заботиться о безопасности применения типов во всем приложении:

```
var distance: any;

// Допускается присваивание значений различных типов
distance = '1000km';
distance = 1000;

// Допускается сочетание различных типов в массиве
var distances: any[] = ['1000km', 1000];
```

Особого упоминания заслуживают литералы `null` и `undefined` из языка JavaScript. Оба они имеют тип `any`. Это позволяет присваивать данные литералы любым переменным, независимо от их типов.

Перечисления

Перечисление – это набор уникальных числовых значений, которым можно присваивать осмысленные имена. Перечисления – это не просто инструмент определения псевдонимов для чисел. Их можно использовать для определения списков вариантов с определенными значениями, в удобной и наглядной форме.

Перечисления объявляются с помощью ключевого слова `enum`, без `var` или любых других спецификаторов, используемых для объявления переменных. Нумерация элементов в них начинается с `0`, если явно не присваиваются иные числовые значения:

```
enum Brands { Chevrolet, Cadillac, Ford, Buick, Chrysler, Dodge };
var myCar: Brands = Brands.Cadillac;
```

Обращение к переменной `myCar` вернет `1` (индекс элемента `Cadillac` в перечислении). Как уже упоминалось, элементам перечисления можно присвоить свои числовые значения:

```
enum BrandsReduced { Tesla = 1, GMC, Jeep };
var myTruck: BrandsReduced = BrandsReduced.GMC;
```

Обращение к переменной myTruck вернет 2, поскольку первому элементу перечисления было присвоено значение 1. Мы можем присвоить значения всем элементам перечисления, лишь бы эти значения были целыми числами:

```
enum StackingIndex {
  None = 0,
  Dropdown = 1000,
  Overlay = 2000,
  Modal = 3000
};
var mySelectBoxStacking: StackingIndex = LayerStackingIndex.Dropdown;
```

И наконец, хочу отметить еще одну хитрость: перечисления позволяют получить имя элемента по соответствующему числовому значению:

```
enum Brands { Chevrolet, Cadillac, Ford, Buick, Chrysler, Dodge };
var myCarBrandName: string = Brands[1];
```

Тип void

Тип void сообщает об отсутствии типа, и его применение ограничивается аннотированием функций, которые не возвращают никакого значения, то есть ничего не возвращают. Этот тип уже использовался в примере из предыдущей главы:

```
resetPomodoro(): void {
  this.minutes = 24;
  this.seconds = 59;
}
```

Автоматическое определение типов

Указание типов данных является необязательным, поскольку TypeScript достаточно умен и способен с определенной степенью точности определять типы переменных и возвращаемых значений функций из контекста. Когда определить тип не удается, TypeScript назначает динамический тип any, сводя контроль типов к минимуму.

Однако старайтесь всегда указывать тип обрабатываемой информации явно, чтобы пользоваться преимуществами проверки корректности всего кода компилятором.

Функции, лямбда-функции и поток выполнения

Так же как в языке JavaScript, функции являются механизмами, которые анализируют входную информацию, упорядочивают ее и применяют необходимые преобразования к данным либо для изменения состояния приложения, либо для возврата вывода, который будет использоваться для формирования логики приложения или интерактивного взаимодействия с пользователем.

Функции в TypeScript ничем не отличаются от обычных функций в JavaScript, за исключением того, что могут аннотироваться статическими типами и таким способом информировать компилятор о том, что они ожидают получить и данные какого типа они собираются вернуть, если таковые имеются.

Аннотации типов в функциях

Следующий пример демонстрирует аннотирование обычной функции в TypeScript:

```
function sayHello(name: string): string {
  return 'Hello, ' + name;
}
```

Сразу можно заметить два основных отличия от обычного синтаксиса функций в JavaScript. Во-первых, в сигнатуре функции присутствует аннотация типа параметра. Это поможет компилятору проверить тип аргумента при вызове функции. В дополнение к этому в объявлении функции указан тип `string` возвращаемого значения. В случаях, когда объявляемая функция не возвращает никакого значения, аннотация типа `void` даст компилятору информацию, необходимую для надлежащей проверки типов.

Как уже упоминалось в предыдущем разделе, компилятор TypeScript достаточно умен, чтобы автоматически определять типы в отсутствие аннотаций. В этом случае компилятор исследует передаваемые аргументы и операторы `return` и на основании этих исследований определит тип возвращаемого значения.

Кроме того, функция в TypeScript может быть представлена как выражение анонимной функции, где объявление функции присваивается переменной:

```
var sayHello = function(name: string): string {
  return 'Hello, ' + name;
};
```

Однако этот способ имеет и обратную сторону. Несмотря на присутствие аннотаций типов в таких выражениях и поддержку автоматического определения типов, компилятор не определяет типов подобных переменных. Можно предположить, что переменная, которая ссылается на функцию, возвращающую значение типа `string`, является строкой. Но в действительности это не так. Переменная, ссылающаяся на анонимную функцию, должна аннотироваться типом функции, который информирует компилятор и о типах аргументов, передаваемых в функцию, и о типе возвращаемого значения, если таковое имеется. Это целый блок в форме: `(аргументы: тип) => тип_возвращаемого_значения`:

```
var sayHello: (name: string) => string = function(name: string): string{
  return 'Hello, ' + name;
};
```

Вы можете спросить: а для чего такой громоздкий синтаксис? Иногда нам требуются переменные, которым могут присваиваться разные функции. Кроме того, всегда желательно стремиться дать компилятору как можно больше сведений. Следующий простой пример поможет лучше понять это:

```
// Две функции с одинаковыми типами, но разной логикой
function sayHello(input: string): string {
  return 'Hello ' + input;
}

function sayHi(input: string): string {
  return 'Hi ' + input;
}

// Здесь объявляется переменная с типом функции
var greetMe: (name: string) => string;

// Затем выполняется присваивание функции переменной
greetMe = sayHello;
```

Таким способом можно гарантировать, что в дальнейшем переменной будут присваиваться только функции, соответствующие аннотации типа в объявлении переменной.

Параметры функций в TypeScript

Из-за проверки типов компилятором функции в TypeScript требуют особого внимания.

Необязательные параметры

Типы параметров в обязательном порядке проверяются компилятором TypeScript. Другими словами, нельзя объявить параметры и затем не включить их в вызов функции. То же относится к передаче аргументов, которые не были объявлены и аннотированны при определении функции. Необходим способ справиться с этой проблемой, и TypeScript предоставляет такую возможность: добавив символ ? в конец имени параметра, такой параметр можно сделать необязательным:

```
function greetMe(name: string, greeting?: string): string {
  if (!greeting) {
    greeting = 'Hello';
  }
  return greeting + ', ' + name;
}
```



Когда параметр отмечен как необязательный и не передается в вызов функции, TypeScript присваивает ему значение `null`. Кроме того, в объявлениях функций сначала перечисляются обязательные параметры, а затем добавляются необязательные.

Параметры по умолчанию

TypeScript предоставляет еще одну возможность, помогающую справиться с ситуацией, описанной выше: параметры со значениями по умолчанию. Мы можем определить значения по умолчанию для некоторых параметров и затем опускать их при вызове функции. Синтаксис очень прост, как показывает модифицированная версия предыдущего примера:

```
function greetMe(name: string, greeting: string = 'Hello'): string {
  return greeting + ', ' + name;
}
```

Про аналогии с необязательными параметрами параметры со значениями по умолчанию должны находиться в объявлениях функций после обязательных параметров.

Оставшиеся параметры

Одним из важных преимуществ, обеспечивающих гибкость языка JavaScript, является возможность принимать в функциях неограниченный массив необъявленных параметров в виде объекта `arguments`. В статически типизированных языках, таких как TypeScript, такое невозможно, но это можно реализовать с помощью объекта оставшихся параметров. Для этого в конец списка аргументов добавляется дополнительный параметр, начинающийся с многоточия и аннотированный типом массива:

```
function greetPeople(greeting: string, ...names: string[]): string {
  return greeting + ', ' + names.join(' and ') + '!';
}

alert(greetPeople('Hello', 'John', 'Ann', 'Fred'));
```



Важно отметить, что оставшиеся параметры должны находиться в конце списка аргументов и могут быть опущены, если не требуются при вызове функции.

Перегрузка функций

Перегрузка методов и функций широко используется в других языках программирования, таких как C#. Однако реализация этой возможности в TypeScript затруднена тем, что в языке JavaScript, на который переводятся программы на TypeScript, отсутствует встроенная поддержка этой возможности. Поэтому единственно возможное решение данной проблемы заключается в объявлении сигнатур для каждой перегруженной версии и последующей реализации общей функции, список параметров и возвращаемое значение которой совместимы с остальными:

```
function hello(name: string): string;
function hello(names: string[]): string;
function hello(names: any, greeting?: string): string {
var namesArray: string[];

if(Array.isArray(names)) {
  namesArray = names;
} else {
  namesArray = [names];
}

if(!greeting) {
  greeting = 'Hello';
```

```
  }

  return greeting + ', ' + namesArray.join(' and ') + '!';
}
```

В этом примере объявлены три перегруженные версии функции, каждая из которых имеет разные аннотации типов. При необходимости можно даже определить разные типы возвращаемых значений. Для этого общая функция должна быть аннотирована типом возвращаемого значения `any`.

Улучшенный синтаксис функций и область обработки с лямбда-функциями

Спецификация ECMAScript 6 вводит понятие стрелочных функций (в других языках программирования, таких как Python, C#, Java или C++, их также называют лямбда-функциями) для упрощения синтаксиса обычных функций, а также для предоставления «пуленепробиваемого» подхода к обработке области видимости функций, традиционная обработка которой печально известна проблемами с ключевым словом `this`.

Первое, что бросается в глаза, – краткий синтаксис, который объясняется тем, что в большинстве случаев стрелочные функции представляют собой однострочные, анонимные выражения:

```
var double = x => x * 2;
```

Это выражение – функция, удваивающая переданное число `x` и возвращая результат, хотя оно не содержит ключевых слов `function` и `return`. Если функция должна принимать несколько аргументов, их список следует заключить в скобки:

```
var add = (x, y) => x + y;
```

Это делает такой синтаксис очень удобным для определения функциональных операций, таких как `map`, `reduce` и др.:

```
var reducedArray = [23, 5, 62, 16].reduce((a, b) => a + b, 0);
```

Стрелочные функции могут содержать инструкции. В этом случае вся реализация заключается в фигурные скобки:

```
var addAndDouble = (x, y) => {
  var sum = x + y;
  return sum * 2;
}
```

А теперь разберемся, какое отношение к области видимости имеет ключевое слово `this`. Вообще, ключевое слово `this` может ссылаться на разные контексты, в зависимости от того, где выполняется функция. Это важно для языка, который гордится превосходной поддержкой функционального стиля программирования, где таким приемам, как обратные вызовы, придается первостепенное значение. Внутри обратного вызова ссылка `this` теряет контекст верхнего уровня, поэтому часто приходится использовать дополнительные соглашения, например присваивать значение `this` переменной `self` или `that`, которая будет использоваться внутри обратного вызова. Идеальным примером могут служить инструкции, содержащие вызовы функций для работы с таймером:

```
function delayedGreeting(name): void {
  this.name = name;
  this.greet = function() {
    setTimeout(function() {
      alert('Hello ' + this.name);
    }, 0);
  }
}

var greeting = new delayedGreeting('Peter')
greeting.greet(); // выведет диалог с текстом 'Hello undefined'
```

Предыдущий фрагмент выведет не требуемый текст `Hello Peter`, а подпорченную строку с досадным приветствием *Mr. Undefined!*. Проблема заключается в запутанной конструкции лексической области видимости `this` при выполнении функции, вызываемой после задержки. Но если перевести этот сценарий на стрелочные функции, мы получим нужный результат:

```
function delayedGreeting(name): void {
  this.name = name;
  this.greet = function() {
    setTimeout(() => alert('Hello ' + this.name), 0);
  }
}
```

Даже если тело стрелочной функции разбить на несколько инструкций, обернув их фигурными скобками, лексическая область видимости ключевого слова `this` останется без изменений, и оно будет ссылаться на надлежащий контекст за пределами вызова `setTimeout`, что обеспечивает более элегантный и наглядный синтаксис.

Классы, интерфейсы и наследование классов

А теперь, после обзора наиболее важных элементов TypeScript, пришло время перейти к знакомству с тем, как все это использовать при создании классов. Классы являются основными строительными блоками приложений, использующих TypeScript и Angular 2.

Хотя `class` и является зарезервированным словом в JavaScript, этот язык никогда не имел фактической реализации традиционных объекто-ориентированных классов, как другие языки, такие как Java или C#. Разработчики на JavaScript могут лишь имитировать такую функциональность, используя объект функции в роли конструктора типа, от которого в дальнейшем создаются экземпляры с помощью оператора `new`. Другими популярными способами расширения объектов функций являются наследование прототипов и композиция.

Теперь появилась настоящая поддержка классов, достаточно гибкая и мощная для реализации приложений. Мы уже видели мельком классы в предыдущей главе, а сейчас рассмотрим их более подробно.

Анатомия классов: конструкторы, свойства и методы

Следующий фрагмент демонстрирует, как выглядит типичный класс. Обратите внимание, что сначала перечисляются свойства класса, затем следуют конструктор, несколько методов и методы доступа к свойствам. При их определении не используется зарезервированное слово `function`, и все свойства и методы аннотируются типами, за исключением конструктора `constructor`:

```
class Car {
  private distanceRun: number = 0;
  color: string;

  constructor(public isHybrid: boolean, color: string = 'red') {
    this.color = color;
  }

  getGasConsumption(): string {
    return this.isHybrid ? 'Very low' : 'Too high!';
  }

  drive(distance: number): void {
    this.distanceRun += distance;
```

```
}

static honk(): string {
  return 'HOOONK!';
}

get distance(): number {
  return this.distanceRun;
}
}
```

Своей организацией этот класс напоминает класс компонента, созданный в *главе 1 «Создание самого первого компонента в Angular 2»*. Вообще говоря, инструкция `class` служит оберткой для нескольких элементов.

- **Члены**: любой экземпляр класса `Car` будет иметь два свойства: строковое свойство `color` и числовое свойство `distanceRun`, доступные только внутри класса. При создании экземпляра этого класса свойство `distanceRun` или любое другое свойство или метод, отмеченное как `private`, не будет доступно как часть прикладного программного интерфейса объекта.
- **Конструктор**: функция-конструктор вызывается непосредственно в момент создания экземпляра класса. Обычно конструктор инициализирует свойства класса значениями своих аргументов. Сигнатуру конструктора можно также использовать для объявления свойств класса, как это было сделано для свойства `isHybrid`. Для этого нужно перед параметром конструктора указать модификатор доступа `private` или `public`. В конструкторах допускается использовать *оставшиеся* и *необязательные* параметры, а также параметры со значениями *по умолчанию*, как это сделано в предыдущем примере с параметром `color`, получающим значение `"red"`, когда он явно не определен.
- **Методы**: методы – это особый вид членов, представляющих функции, которые возвращают или не возвращают типизированное значение. Обычно эти функции становятся частью прикладного программного интерфейса объекта. Методы могут быть и закрытыми. В этом случае они используются внутри класса как вспомогательные функции для поддержки других методов класса.
- **Статические члены**: члены, отмеченные как `static`, связаны с классом, а не с экземплярами объектов этого класса. То есть

их можно использовать напрямую, без создания экземпляра объекта. Более того, статические члены недоступны из экземпляров объектов и сами не могут обращаться к другим членам класса с помощью ключевого слова `this`. Обычно они включаются в определение класса как вспомогательные или фабричные методы, обеспечивающие универсальную функциональность, не относящиеся к конкретному экземпляру объекта.

- **Методы доступа к свойствам**: пользовательские методы доступа к свойствам в ES5 определяются с использованием громоздкого и неудобного синтаксиса `Object.defineProperty`. Теперь это делается проще. Чтобы реализовать метод доступа к свойству (обычно к закрытому, как в предыдущем примере), необходимо добавить метод, имя которого начинается с префикса `set` (для записи) или `get` (для чтения), за которым следует имя свойства.

Почему бы в качестве упражнения не скопировать предыдущий фрагмент кода на страницу для экспериментов (http://www.typescriptlang.org/Playground) и не опробовать его? Можно даже увидеть объект экземпляра класса `Car` в действии, добавив следующий фрагмент кода сразу после определения класса, и, запустив его, просмотреть вывод в консоль с помощью браузерных инструментов разработчика:

```
var myCar = new Car(false);

console.log(myCar.color);    // 'red'
// Открытый метод доступа вернет distanceRun:
console.log(myCar.distance); // 0

myCar.drive(15);
console.log(myCar.distance); // 15 (0 + 15)
myCar.drive(21);
console.log(myCar.distance); // 36 (15 + 21)

// Каков выхлоп автомобиля этого типа?
myCar.getGasConsumption();   // 'Too high!'

Car.honk(); // 'HOOONK!' создавать экземпляр объекта не нужно
```

Можно даже выполнить дополнительное тестирование, добавив следующие недопустимые инструкции, пытающиеся получить доступ к закрытому свойству `distanceRun` и установить значение свойства `distance`, не имеющего метода записи.

```
console.log(myCar.distanceRun);
myCar.distance = 100;
```

Сразу после вставки этих инструкций в поле ввода на странице для экспериментов они будут подчеркнуты красным, что отражает их неправильность. Тем не менее можно направить этот код на компиляцию и запуск, так как ES5 позволяет это. В итоге, если попытаться скомпилировать этот файл, среда выполнения выведет следующие сообщения об ошибках:

```
example 26.ts(21,7): error TS1056: Accessors are only available when
targeting ECMAScript 5 and higher (Методы доступа применяются только при ис-
пользовании  ECMAScript 5 и выше).

example_26.ts(29,13): error TS2341: Property 'distanceRun' is private and
only accessible within class 'Car' (Свойство 'distanceRun' является закрытым
и доступно только внутри класса 'Car').
```

Интерфейсы в TypeScript

С увеличением масштаба приложения и количества классов и конструкций в нем остро встает проблема обеспечения согласованности и соблюдения правил в коде. Одним из лучших способов решения проблемы согласованности и контроля допустимости типов является создание интерфейсов.

По своей сути интерфейс – это макет, определяющий конкретную схему полей и любых типов (классов или функций), реализующих этот интерфейс. Интерфейсы весьма полезны при применении строгой типизации к классам, генерируемым фабриками, когда сигнатуры функций определяются для уверенности, что заданное типизированное свойство содержит нужные данные, или в других ситуациях.

Давайте приступим к делу! Определим интерфейс транспортного средства `Vehicle`. Интерфейс `Vehicle` не является классом, это схема, которой должен соответствовать любой класс, реализующий ее:

```
interface Vehicle {
  make: string;
}
```

Любой класс, реализующий интерфейс `Vehicle`, должен иметь свойство `make` типа `string`, согласно приведенному выше примеру. В противном случае компилятор TypeScript будет недоволен:

```
class Car implements Vehicle {
  // Компилятор выдаст предупреждение, если не определить свойство
     make: string;
}
```

Как следствие интерфейсы удобно использовать для определения минимального набора свойств и методов нужного типа и согласования всей базы кода.



Важно отметить, что интерфейсы используются не только для определения минимальных схем классов, но и для других типов. То есть с помощью интерфейсов можно гарантировать присутствие определенных методов в классах и свойств в объектах, используемых в дальнейшем как параметры функций, типы функций, типы элементов в массивах и даже переменных. Интерфейс может содержать необязательные члены, а также члены, типы которых определяются другими интерфейсами.

Приведем пример. Используем префикс `I` (в верхнем регистре) в именах всех видов интерфейсов. Так будет легче определять типы при использовании функции автозавершения кода в интегрированных средах разработки.

Сначала определим интерфейс `IException`, определяющий схему типа с обязательным свойством `message` и необязательным числовым свойством `id`:

```
interface IException {
  message: string;
  id?: number;
}
```

Имеется возможность определять интерфейсы и для элементов массивов. Для этого следует определить интерфейс с единственным членом, определяющим индекс как число или строку (для словарей) и тип элементов массива. В данном случае нам требуется создать интерфейс для массивов, элементы которых имеют тип `IException`. Этот тип содержит строковое свойство `message` и необязательное числовое свойство `id`, как уже упоминалось в предыдущем примере:

```
interface IExceptionArrayItem {
  [index: number]: IException;
}
```

А теперь определим основу будущего класса с помощью типизированного массива и метода с определенным возвращаемым типом:

```
interface IErrorHandler {
  exceptions: IExceptionArrayItem[];
  logException(message: string, id?: number): void;
}
```

Интерфейсы можно также определять для отдельных типов объектов. Это очень удобно для определения шаблонов конструктора и методов, как показано ниже:

```
interface IExceptionHandlerSettings {
  logAllExceptions: boolean;
}
```

Наконец, следующий класс реализует все перечисленные типы интерфейсов:

```
class ErrorHandler implements IErrorHandler {
  exceptions: IExceptionArrayItem[];
  logAllExceptions: boolean;

  constructor(settings: IExceptionHandlerSettings) {
    this.logAllExceptions = settings.logAllExceptions;
  }

  logException(message: string, id?: number): void {
    this.exceptions.push({ message, id });
  }
}
```

Фактически мы определили класс обработки ошибок, который управляет внутренним массивом исключений и экспортирует метод для сохранения новых исключений в упомянутый выше массив. Эти два элемента определяются с помощью интерфейса `IErrorHandler` и являются обязательными. Конструктору класса требуются параметры, определенные в интерфейсе `IExceptionHandlerSettings`, он использует их при заполнении массива элементами, типизированными как `IException`. Попытка создать экземпляр класса `ErrorHandler` без параметра `logAllExceptions` вызовет сообщение об ошибке.

Завершим раздел об интерфейсах, подчеркнув, что классы могут реализовывать несколько интерфейсов.

Расширение классов с помощью наследования

Подобно тому, как класс может быть определен с помощью интерфейса, он может также наследовать свойства и методы других классов и использовать их, как если бы они были его собственными. Чтобы унаследовать свойства и методы другого класса, нужно добавить ключевое слово `extends` к имени вновь определяемого класса и указать имя наследуемого класса:

```
class Sedan extends Car {
  model: string;
  constructor(make: string, model: string) {
    super(make);
    this.model = model;
  }
}
```

Здесь происходит наследование родительского класса `Car`, который обладает свойством `make`. Мы можем устанавливать значения свойств, объявленных в родительском классе, и даже вызывать его конструктор с помощью метода `super()`, который ссылается на конструктор родительского класса. Также можно переопределять методы родительского класса, добавляя свои методы с теми же именами. При этом сохраняется возможность вызова оригинальных методов родительского класса через объект `super`. А теперь вновь обратимся к интерфейсам: они тоже могут наследовать определения других интерфейсов. Проще говоря, интерфейс может наследовать другой интерфейс.



Важно отметить, что ES6 и TypeScript не поддерживают множественного наследования. Поэтому совместить функциональность из разных источников можно лишь посредством составных или посреднических классов.

Декораторы TypeScript

Декораторы – очень удобный инструмент – первоначально появились в Google **AtScript** (надмножество TypeScript, которое окончательно было включено в TypeScript в начале 2015 г.) и являются частью текущей спецификации ECMAScript 7. По сути, декораторы определяют способ добавления метаданных в объявления классов для использования их механизмом внедрения зависимостей или директивами компиляции (http://blogs.msdn.com/b/somasegar/archive/2015/03/05/typescript-lt-3-angular.aspx). Путем создания декораторов определяются специальные аннотации, способные влиять на поведения классов, методов или функций или просто изменять данные, которые были определены в полях или в параметрах. В этом смысле декораторы являются мощным средством расширения оригинальных типов без создания подклассов и наследования.

Они, безусловно, одна из наиболее интересных особенностей TypeScript, широко используются в Angular 2 при разработке директив и компонентов, а также для управления внедрением зависимостей,

как будет показано в *главе 4 «Расширение компонентов с помощью директив и фильтров»*.



Декораторы легко узнаваемы по префиксу @ перед их именами и обычно располагаются над элементами, которые они *декорируют*, независимо от содержания этих элементов.

Можно определить до четырех разных видов декораторов, в зависимости от декорируемого элемента:

- декораторы классов;
- декораторы свойств;
- декораторы методов;
- декораторы параметров.

Рассмотрим каждый из них!

Декораторы классов

Декораторы классов позволяют расширить класс или выполнить операции над любым из его свойств и методов. Декоратор выполняется перед созданием экземпляра класса.

Чтобы создать декоратор класса, требуется определить обычную функцию, принимающую указатель на конструктор декорируемого класса – параметр типа `Function` (или любого другого типа, наследующего тип `Function`). Формальное объявление `ClassDecorator` выглядит следующим образом:

```
declare type ClassDecorator = <TFunction extends Function>(Target: TFunction)
=> TFunction | void;
```

Да, действительно трудно понять, что означает этот бред, верно? Давайте поместим все это в контекст простого примера:

```
function Greeter(target: Function): void {
  target.prototype.greet = function(): void {
    console.log('Hello!');
  }
}

@Greeter
class Greeting {
  constructor() {
    // Здесь находится реализация...
  }
}
```

```
var myGreeting = new Greeting();
myGreeting.greet();   // в консоль будет выведено 'Hello!'
```

Как видите, декоратор `Greeter` добавил в класс `Greeting` метод `greet()`, который отсутствовал в нем изначально.

Расширение сигнатуры функции декоратором класса

Иногда требуется изменить порядок работы декоратора с помощью дополнительных параметров. Нет ничего проще! Мы можем создавать декораторы с произвольной сигнатурой, главное, чтобы они возвращали функцию с той же сигнатурой, как в случае создания декоратора без параметров. Чтобы создать декоратор с параметрами, достаточно просто определить функцию, сигнатура которой соответствует количеству и типам параметров декоратора. Эта функция должна возвращать другую функцию, сигнатура которой соответствует определяемому декоратору.

Следующий фрагмент демонстрирует ту же функциональность, что и предыдущий пример, но позволяет разработчикам менять приветствие:

```
function Greeter(greeting: string) {
  return function(target: Function) {
    target.prototype.greet = function(): void {
      console.log(greeting);
    }
  }
}

@Greeter('Howdy!')
class Greeting {

  constructor() {
    // Здесь находится реализация...
  }
}

var myGreeting = new Greeting();
myGreeting.greet();   // в консоль будет выведено 'Howdy!'
```

Декораторы свойств

Декораторы свойств применяются к полям классов и определяются путем создания функции `PropertyDecorator`, чья сигнатура содержит два параметра:

- `target`: прототип декорируемого класса;
- `key`: имя декорируемого свойства.

Декораторы этого вида можно использовать, например, для журналирования значений, присваиваемых полям класса при создании экземпляров, или даже для определения реакции на изменение полей. Рассмотрим пример, реализующий обе эти возможности:

```
function LogChanges(target: Object, key: string) {
  var propertyValue: string = this[key];

  if (delete this[key]) {
    Object.defineProperty(target, key, {
      get: function() {
        return propertyValue;
      },
      set: function(newValue) {
        propertyValue = newValue;
        console.log(`${key} is now ${propertyValue}`);
      }
    });
  }

  class Fruit {
    @LogChanges
    name: string;
  }

  var fruit = new Fruit();
  fruit.name = 'banana';   // выведет в консоль 'name is now banana'
  fruit.name = 'plantain'; // выведет в консоль 'name is now plantain'
```

Здесь применима та же логика параметризации, что и для декораторов классов, хотя сигнатура возвращаемой функции несколько отличается, чтобы соответствовать объявлению декоратора без параметров.

Следующий пример демонстрирует, как реализовать журналирование изменений свойства класса с вызовом пользовательской функции-обработчика:

```
function LogChanges(callbackObject: any): Function {
  return function(target: Object, key: string): void {
    var propertyValue: string = this[key];
    if (delete this[key]) {
      Object.defineProperty(target, key, {
```

```
    get: function() {
      return propertyValue;
    },
    set: function(newValue) {
      propertyValue = newValue;
      callbackObject.onchange.call(this, propertyValue);
    }
  });
    }
  }
}

class Fruit {
  @LogChanges({
    onchange: function(newValue: string): void {
      console.log(`The fruit is ${newValue} now`);
    }
  })
  name: string;
}

var fruit = new Fruit();
fruit.name = 'banana';   // консоль: 'The fruit is banana now'
fruit.name = 'plantain'; // консоль: 'The fruit is plantain now'
```

Декораторы методов

Эти специальные декораторы способны обнаруживать, журналировать и влиять на выполнение методов. Для этого требуется определить функцию `MethodDecorator`, принимающую следующие параметры:

- `target`: объект, представляющий декорируемый метод;
- `key`: строка с именем декорируемого метода;
- `value`: дескриптор свойства данного метода. Фактически это объект хэша, содержащий среди прочего свойство `value` со ссылкой на сам метод.

Рассмотрим использование функции `MethodDecorator` на примере. Предположим, что требуется создать многоцелевой регистратор, независимый от сигнатур, который будет отслеживать результаты, возвращаемые каждым методом класса, включая некоторые дополнительные данные, например время выполнения метода:

```
function LogOutput(target: Function, key: string, descriptor: any) {
  var originalMethod = descriptor.value;
  var newMethod = function(...args: any[]): any {
```

```
    var result: any = originalMethod.apply(this, args);
    if(!this.loggedOutput) {
      this.loggedOutput = new Array<any>();
    }
    this.loggedOutput.push({
      method: key,
      parameters: args,
      output: result,
      timestamp: new Date()
    });
    return result;
  };

  descriptor.value = newMethod;
}
```

Как уже упоминалось, параметр `descriptor` содержит ссылку на декорируемый метод, и ничто не мешает заменить ее ссылкой на свой метод. Мы можем воспользоваться этим, выполнив вызов оригинального метода во вновь созданном, с передачей ему все тех же параметров.



Не забывайте, что декораторы методов находятся внутри класса, представленного в параметре `target`, что позволяет расширить класс собственными свойствами и методами. Но будьте внимательны, поскольку есть риск переопределить уже существующие свойства и методы. В этом примере отсутствуют связанные с этим меры предосторожности, но вы должны позаботиться о них при написании своего кода.

Как видите, код получился простым и наглядным. Мы сохранили ссылку на оригинальный метод из `descriptor.value` в переменной `originalMethod`. Затем создали новый метод с переменным списком аргументов, что гарантирует совместимость с сигнатурами любых методов. Внутри нового метода вызываем оригинальный и сохраняем результат (вместе с другими данными, такими как имя вызванного метода, использованные параметры, время и дату вызова) в поле-массиве класса, которое создается автоматически. Затем возвращаем результат вызова оригинального переопределенного метода.

Давайте протестируем наш впечатляющий декоратор! Создадим класс с методом, производящим вычисления, добавим журналирование и посмотрим, что из этого получится:

```
class Calculator {
  @LogOutput
  double (num: number): number {
```

```
    return num * 2;
  }
}

var calc = new Calculator();
calc.double(11);

console.log(calc.loggedOutput); // Выведет массив [Object] в консоль
```

Класс `Calculator` экспортирует метод, удваивающий любое число, переданное ему в виде аргумента. Но журналируются ли выполняемые операции? Исследуйте значение `calc.loggedOutput` в консоли браузера, и вы увидите, что произошло.

Можно попробовать добавить другой метод, с совершенно другим типом и сигнатурой, чтобы убедиться, что декоратор продолжает нормально работать. Добавьте его в тело класса `Calculator`:

```
@LogOutput
doNothing (input: any): any {
  return input;
}
```

А теперь выполните следующую инструкцию в консоли браузера:

```
calc.doNothing(['Learning Angular 2', 2016]);
```

Если теперь вывести значение `calc.loggedOutput`, можно увидеть, что наряду с информацией о предыдущих вычислениях в массиве появился новый объект.

Следующая строка в примере должна была привлечь ваше внимание:

```
this.loggedOutput = new Array<any>();
```

Создавая новое свойство в классе, требуется указать его тип. Но, так как здесь нельзя применить обычное аннотирование, был использован обобщенный конструктор объекта `Array`, параметризованный типом `any` в угловых скобках.

В этой главе мы не будем рассматривать приемы обобщенного программирования (Generics), поскольку это выходит за рамки такой вводной книги, как эта. Поэтому за более подробной информацией обращайтесь на официальный сайт (http://www.typescriptlang.org/docs/handbook/generics.html[1]). А пока просто имейте в виду, что обобщенное программирование позволяет применить конкретный

---

[1] http://typescriptrus.blogspot.ru/2015/10/6.html. – *Прим. ред.*

тип при выполнении определенных пользовательских методов или использовании специфических классов, таких как массивы. В дальнейшем вы увидите, как приемы обобщенного программирования помогают организовать поддержку строго типизированных массивов или вновь созданных пустых объектов в интерфейсах. И снова за дополнительной информацией обращайтесь на официальный сайт языка TypeScript.

Декораторы параметров

Последний вид декораторов – `ParameterDecorator` – используется для декорирования параметров в сигнатурах функций. Декоратор этого вида не предназначен ни для изменения сведений о параметрах, ни для смены поведения функции, но он получает значения параметров и может выполнять дополнительные операции, например журналировать или сохранять данные. Функция `ParameterDecorator` принимает следующие параметры:

- `target`: прототип объекта с функцией, чьи параметры декорируются;
- `key`: имя функции с декорируемым параметром;
- `parameterIndex`: индекс параметра, к которому применяется декорирование, в массиве параметров.

Следующий пример демонстрирует использование декоратора параметров:

```
function Log(target: Function, key: string, parameterIndex: number) {
  var functionLogged = key || target.prototype.constructor.name;
  console.log(`
    The parameter in position ${parameterIndex} at
    ${functionLogged} has  been decorated
  `);
}

class Greeter {
  greeting: string;
  constructor(@Log phrase: string) {
    this.greeting = phrase;
  }
}

// В консоль будет выведено то, что определено в классе выше:
// 'The parameter in position 0 at Greeter has been decorated'
```

Возможно, вы обратили внимание на необычное выражение, результат которого присваивается переменной `functionLogged`. Дело

в том, что значение параметра target меняется в зависимости от вида функции, параметры которой декорируются. Для конструктора возвращается ссылка на прототип класса, а для метода – ссылка на функцию конструктора. То же относится и к параметру key, который получает значение undefined при декорировании параметра конструктора.

Как уже упоминалось в начале этого раздела, декораторы параметров не предназначены для изменения их значений или поведения методов и конструкторов, содержащих эти параметры. Их целью обычно является журналирование или *подготовка* объекта для реализации дополнительных уровней абстракции или возможностей посредством декораторов более высокого уровня, таких как декораторы методов или классов.

Типичные сценарии применения декораторов для журналирования поведения компонента или управления внедрением зависимостей будут рассмотрены в *главе 4 «Расширение компонентов с помощью директив и фильтров»*.

Работа с модулями в приложении

С увеличением размера приложения приходит время, когда требуется улучшенная организация кода, позволяющая сделать его устойчивым и обеспечить его повторное использование. Эта потребность обеспечивается модулями, поэтому рассмотрим механизм их работы и порядок их применения в приложении. Модули могут быть внутренними или внешними. В этой книге основное внимание уделяется внешним модулям, тем не менее ниже приводится обзор модулей обоих типов.

Внутренние модули

Внутренние модули по своей сути являются простыми оболочками, содержащими набор классов, функций, объектов или переменных, находящихся во внутренней, а не в глобальной или внешней области видимости. Чтобы экспортировать содержимое модуля, нужно перед каждым элементом, который должен быть доступен извне, добавить ключевое слово export:

```
module Greetings {

  export class Greeting {
    constructor(public name: string) {
      console.log(`Hello ${name}`);
    }
```

```
  }
  export class XmasGreeting {
    constructor(public name: string) {
      console.log(`Merry Xmas ${name}`);
    }
  }
}
```

Модуль `Greetings` содержит два класса, которые становятся доступными за пределами модуля при их импортировании:

```
import XmasGreeting = Greetings.XmasGreeting;
var xmasGreeting = new XmasGreeting('Joe');
// в консоль будет выведено 'Merry Xmas Joe'
```

Просматривая приведенный выше код, можно заключить, что внутренние модули хорошо подходят для группировки и инкапсуляции элементов в контексте пространств имен. Имеется даже возможность разбивать модули на несколько файлов, для чего в каждом файле достаточно объявить одно и то же имя модуля. Но в этом случае требуется определить ссылки на файлы, по которым разбросаны объекты, принадлежащие этому модулю, с помощью тегов `reference`:

```
/// <reference path="greetings/XmasGreeting.ts" />
```

Основным недостатком внутренних модулей является необходимость размещения их всех в одном файле или области видимости приложения, если они располагаются вне области видимости интегрированной среды разработки. Сгенерированные JavaScript-файлы можно подключить в веб-странице с помощью утилиты сборки проектов, такой как Grunt или Gulp, или использовать флаг `--outFile` компилятора TypeScript, чтобы собрать при компиляции все файлы `.ts`, обнаруженные компилятором в рабочей области, в один пакет, с помощью загрузочного файла со ссылочными тегами для всех других модулей:

```
tsc --outFile app.js module.ts
```

Эта команда скомпилирует все файлы TypeScript вместе с файлами зависимостей, на которые указаны ссылки в виде тегов `reference`. Если ссылка на какой-то файл отсутствует, он не будет включен в окончательную сборку, поэтому можно пойти другим путем и указать все файлы, содержащие автономные модули, в команде компиляции или просто добавить файл `.txt` с полным списком подключаемых модулей. Как вариант можно просто использовать внешние модули.

Внешние модули

Внешние модули являются хорошим выбором, когда речь идет о разработке быстро растущих приложений. Внешние модули организованы на уровне файлов, когда каждый файл является модулем и имя модуля соответствует имени файла без расширения `.js`. В этом случае не используется ключевое слово `module` и каждый элемент с префиксом `export` становится частью прикладного программного интерфейса внешнего модуля. Внутренний модуль, представленный в предыдущем примере, легко можно превратить во внешний, хранящийся в файле `Greetings.ts`:

```
export class Greeting {
  constructor(public name: string) {
    console.log(`Hello ${name}`);
  }
}

export class XmasGreeting {
  constructor(public name: string) {
    console.log(`Merry Xmas ${name}`);
  }
}
```

Ниже показано, как импортировать такой модуль и использовать его классы:

```
import greetings = require('Greetings');
var XmasGreetings = greetings.XmasGreetings();
var xmasGreetings = new XmasGreetings('Pete');
// выведет в консоль: 'Merry Xmas Pete'
```

Очевидно, что функция `require` не поддерживается традиционным JavaScript, поэтому нужно указать компилятору, как реализовать ее в целевых JavaScript-файлах. К счастью, компилятор TypeScript имеет параметр `--module`, поэтому можно настроить загрузчик зависимостей, выбрав для проекта: `commonjs` – для импорта в стиле node; `amd` – для импорта в стиле RequireJS; `umd` – для импорта в соответствии со спецификацией Universal Module Definition или `system` – для импорта в стиле SystemJS. В этой книге используется загрузчик SystemJS:

```
tsc --outFile app.js --module commonjs
```

Полученный файл будет оформлен должным образом, и модули смогут загружать зависимости с помощью выбранного нами загрузчика.

Предстоящий путь

Глава подходит к концу. Однако это было лишь введение в TypeScript. Его огромный потенциал невозможно описать в таком небольшом введении, поэтому осталась масса незатронутых вопросов, которые стоит изучить и детально проанализировать. К сожалению, это выходит за рамки книги, и данная информация не требуется для освоения Angular 2, хотя она, безусловно, расширит ваши возможности. Рекомендуем для начала ознакомиться со следующими аспектами спецификации языка:

- обобщенное программирование;
- объекты-смеси (mixin);
- пересечение типов;
- объединение типов;
- кортежи;
- создание файлов с объявлениями типов.

Желающим ближе познакомиться с TypeScript можно порекомендовать книги: «TypeScript Essentials» Кристофера Нэнси (Christopher Nance); «Learning TypeScript» Ремо Х. Джансена (Remo H. Jansen) или более продвинутую «Mastering TypeScript» Натана Розенталса (Nathan Rozentals), изданные Packt Publishing.

Итоги

Глава получилась длинной, но введение в TypeScript было абсолютно необходимо для понимания логики наиболее выдающихся аспектов Angular 2. Она дает возможность не только познакомиться с синтаксисом языка, но и объясняет причину его выбора для создания фреймворка Angular 2. Были рассмотрены особенности архитектуры типов, описаны приемы создания современной бизнес-логики, проектирования функций с широким спектром альтернатив для параметризованных сигнатур и даже способ применения новых мощных стрелочных функций для обхода проблем с областью видимости. Наиболее важной частью этой главы стал обзор классов, методов, свойств, наследования и улучшения структуры приложений с помощью интерфейсов. Другими важными возможностями, рассмотренными в этой главе, стали модули и декораторы, которые, как очень скоро можно будет убедиться, имеют первостепенное значение для понимания механизма внедрения зависимостей в Angular 2.

Имея в своем распоряжении эти знания, можно продолжить исследование фреймворка Angular 2 и уверенно переходить к таким аспектам создания компонентов, как инкапсуляция стилей, форматирование вывода и т. д. *Глава 3 «Реализация свойств и событий компонентов»* знакомит с методами создания продвинутых шаблонов и привязки данных, а также с директивами и фильтрами. Все эти возможности позволят применить на практике недавно полученные знания о языке TypeScript.

Глава 3

Реализация свойств и событий компонентов

Выше был представлен лишь общий обзор процесса включения компонентов в новую экосистему Angular, их роли и поведения в ней, инструментов, необходимых для создания собственных компонентов. Кроме того, был рассмотрен язык TypeScript, оказавшийся идеальным компаньоном и имеющий все необходимое для освоения возможностей, привносимых Angular 2 в создание интерактивных компонентов.

В этой главе мы:

- познакомимся со всеми доступными синтаксическими возможностями привязки данных в шаблонах;
- узнаем, как определять общедоступные API для наших компонентов, позволяющие пользоваться их свойствами и обработчиками событий;
- освоим привязку данных в Angular 2;
- увидим, как упростить управление стилями CSS с помощью инкапсуляции представлений.

Улучшенный синтаксис шаблонов

В *главе 1 «Создание самого первого компонента в Angular 2»* мы рассмотрели процесс встраивания HTML-шаблонов в компоненты, но это было лишь поверхностное знакомство с разработкой шаблонов в Angular 2. Как будет показано ниже, реализация шаблонов тесно

связана с архитектурой теневой модели DOM, и имеется масса удобных синтаксических конструкций, упрощающих задачу декларативной привязки свойств и событий к представлениям.

Компоненты Angular способны экспортировать свойства и методы для взаимодействий с другими компонентами или контейнерами. К их числу относятся входные свойства, используемые для передачи данных в компонент, и выходные свойства, которые можно связать приемниками событий для передачи оперативной информации об изменениях в состоянии компонента.

Давайте посмотрим, как Angular решает проблему ввода/вывода данных в компонентах на простых и наглядных примерах. Постарайтесь сосредоточиться на сути этих возможностей. В работе они будут продемонстрированы позже, по мере доработки проекта Pomodoro.

Привязка данных с помощью входных свойств

Вернемся к проекту Pomodoro из *главы 1 «Создание самого первого компонента в Angular 2»* и представим, что нам нужно добавить в компонент настраиваемый атрибут, позволяющий увеличить или уменьшить время обратного отсчета:

```
<pomodoro-timer [seconds]="25"></pomodoro-timer>
```

Обратите внимание, что атрибут заключен в скобки. Они сообщают фреймворку Angular, что атрибут является входным свойством. Класс, моделирующий компонент `pomodoro-timer`, будет содержать функцию записи в свойство `seconds`, реагирующую на изменение этого значения изменением продолжительности обратного отсчета. Значение можно передать в переменной или определить явно. В последнем случае его следует заключить в одинарные кавычки внутри двойных кавычек, чтобы указать, что значение является текстовой строкой.

В одних случаях такой синтаксис применяется для внедрения данных в свойства компонента, в других – чтобы заставить обычные HTML-атрибуты реагировать на значение полей компонента, как показано ниже:

```
<h1 [hidden]="hideMe">
  Этот текст не виден, если 'hideMe' имеет значение true
</h1>
```

Дополнительные синтаксические конструкции для привязки выражений

Команда Angular реализовала дополнительные синтаксические конструкции, упрощающие выполнение типичных преобразований в директивах компонентов и элементах DOM, таких как настройка имен атрибутов и классов и применение стилей. Ниже приводится несколько примеров таких конструкций, позволяющих значительно сэкономить время за счет декларативного определения привязок к свойствам:

```
<div [attr.hidden]="isHidden">...</div>
<input [class.is-valid]="isValid">
<div [style.width.px]="myWidth">...</div>
```

В первом примере в блок `div` добавляется атрибут `hidden`, если выражение `isHidden` возвращает значение `true`. Кроме логических, можно использовать значения любых других типов, например строковые. Во втором примере в атрибут `class` будет добавлен класс `is-valid`, если выражение `isValid` вернет значение `true`. В третьем примере в блок `div` будет добавлен атрибут `style` со свойством `width`, принимающим значение выражения `myWidth` в пикселях. Другие подобные примеры синтаксических конструкций Angular 2 можно найти в шпаргалке (https://angular.io/cheatsheet) на официальном сайте Angular.

Привязка событий к выходным свойствам

Допустим, что нам требуется организовать отправку из компонента Pomodoro уведомлений об окончании отчета времени, чтобы выполнить определенные действия вне компонента. Этого можно добиться с помощью выходного свойства:

```
<pomodoro-timer (countdownComplete)="onCountownCompleted()">
</pomodoro-timer>
```

Обратите внимание, что на этот раз атрибут заключен в круглые скобки. Они сообщают фреймворку Angular, что такой атрибут является выходным свойством, запускающим подключенный к нему обработчик событий. В этом случае требуется создать обработчик события `onCountownCompleted` в контейнерном объекте, обертывающем данный компонент.



Кстати, применение верблюжьей нотации здесь не случайно. Такое соглашение применяется ко всем именам выходных и входных свойств в Angular 2.

Для многих известных интерактивных событий, таких как `click`, `mouseover`, `mouseout`, `focus` и многих других, уже имеются соответствующие выходные свойства.

```
<button (click)="doSomething()">Click me</button>
```

Входные и выходные свойства в действии

Лучший способ освоить концепции, описанные в предыдущих разделах, – это применить их на практике. Продемонстрируем их на примере таймера Pomodoro из *главы 1 «Создание самого первого компонента в Angular 2»* и разберем этот простой пример. Откройте файл `pomodoro-timer.ts` и измените его содержимое, как показано ниже:

```
import { Component } from '@angular/core';
import { bootstrap } from '@angular/platform-browser-dynamic';
@Component({
  selector: 'countdown',
  template: '<h1>Time left: {{seconds}}</h1>'
})
class CountdownComponent {
  seconds: number = 25;
  intervalId: number;

  constructor() {
    this.intervalId = setInterval(() => this.tick(), 1000);
  }

  private tick(): void {
    if (--this.seconds < 1) {
      clearInterval(this.intervalId);
    }
  }
}
```

Отлично! Мы только что определили простой, но весьма эффективный компонент таймера, способный отсчитывать интервалы времени от 0 до 25 секунд. (Обратили внимание на поле `seconds`? TypeScript поддерживает инициализацию свойств при их объявлении.) Обычный цикл `setInterval` многократно вызывает закрытую пользовательскую функцию `tick()`, уменьшающую значение `seconds`, пока оно не достигнет нуля.

Обратите внимание, что мы не включили вызов функции `bootstrap` для создания экземпляра компонента в HTML-документе, даже при том, что импортировали ее. Это весьма распространенная практика,

потому что функция `bootstrap` вызывается только для создания экземпляра родительского компонента верхнего уровня (или корневого компонента), а все другие компоненты, определяемые как дочерние компоненты, создаются автоматически.

Однако теперь мы должны встроить этот компонент куда-то. Поэтому создадим еще один компонент, который будет служить лишь HTML-оберткой для предыдущего компонента. Сразу после класса `CountdownComponent`, в том же файле, добавьте новый компонент:

```
@Component({
  selector: 'pomodoro-timer',
  directives: [CountdownComponent],
  template: '<countdown></countdown>'
})
class PomodoroTimerComponent {}

bootstrap(PomodoroTimerComponent);
```

Как видите, мы добавили в инициализацию компонента новое свойство `directives` и объявили компонент `CountdownComponent`. Компоненты в Angular 2 в основном являются директивами с шаблонами представлений. Но можно также увидеть директивы без представлений, которые добавляют новые функциональные возможности во вмещающие элементы или просто служат обертками для других элементов. Кроме того, они могут предоставлять дополнительную функциональность другим компонентам через свой прикладной программный интерфейс.

Более подробно директивы рассматриваются в следующей и еще в нескольких главах книги. Сейчас же просто отметим, что когда определяемый компонент содержит другие компоненты, что и было только что реализовано, необходимо явно объявить классы непосредственных дочерних компонентов в параметре `directives` в виде массива. Вам, наверное, интересно, почему принимающий, или родительский, компонент `PomodoroTimerComponent` создан без реализации. Мы добавим в него некоторые возможности ниже, а сейчас используем его как иллюстрацию концепции исходной точки дерева компонентов.

Декларативное определение пользовательских значений

Вы, вероятно, согласитесь с тем фактом, что возможность настройки пользовательских таймеров оказалась совсем нелишней, верно?

Входные свойства являются отличным способом для достижения этой цели. Для использования данной возможности нужно настроить инструкцию импорта в начале файла:

```
import { Component, Input } from '@angular/core';
```

и немного изменить сам таймер:

```
@Component({
  selector: 'countdown',
  template: '<h1>Time left: {{seconds}}</h1>'
})
class CountdownComponent {
  @Input() seconds: number;
  intervalId: number;
  // Остальная реализация осталась без изменений...
}
```

Как видите, теперь поле `seconds` не инициализируется и декорируется декоратором свойства (который рассматривался в *главе 2 «Введение в TypeScript»*). Это только начало определения программного интерфейса компонента.



Регистр символов в именах свойств имеет значение, а соглашение об именовании в Angular 2 требует применения верблюжьей нотации к именам входных свойств компонентов и, как будет продемонстрировано ниже, к именам выходных свойств тоже.

Затем нужно просто добавить необходимое свойство в шаблон контейнерного компонента:

```
@Component({
  selector: 'pomodoro-timer',
  directives: [CountdownComponent],
  template: `<div class="container text-center">
              <img src="assets/img/pomodoro.png" />
              <countdown [seconds]="25"></countdown>
            </div>`
})
```

Обратите внимание, что в компонент `PomodoroTimerComponent` не внесено никаких изменений. Мы изменили лишь его дочерний компонент `CountdownComponent`. Однако его новый программный интерфейс доступен любому компоненту, который включит его в свой шаблон в качестве дочернего компонента, поэтому его свойства можно настраивать декларативно, прямо из шаблона, или, при желании, даже

привязать его значение императивно, из свойства, находящегося в классе контроллера `PomodoroTimerComponent`.



Декоратор `@Input()` позволяет определить псевдоним для декорируемого свойства, под которым оно будет доступно в HTML. Для этого нужно просто указать выбранное имя в декораторе, например: `@Input('имя_свойства')`. Однако такая практика не поощряется, поскольку экспортирование свойств под именами, отличающимися от тех, что определены в классе контроллера, может привести к путанице.

Взаимодействие между компонентами через пользовательские события

А теперь, после настройки дочернего компонента, как передать сообщение от дочернего компонента к родительскому? Здесь приходят на помощь пользовательские события! Для передачи событий необходимо определить в дочернем компоненте выходное свойство и связать его с функцией-обработчиком в родительском компоненте.

С этой целью импортируем класс `EventEmitter` и декоратор `@Output`, предназначение которого обратно уже рассмотренному ранее декоратору `@Input`:

```
import { Component, Input, Output, EventEmitter } from '@angular/ core';
```

Класс `EventEmitter` является встроенным средством передачи событий в Angular 2. По сути, `EventEmitter` реализует отправку уведомлений об изменении отслеживаемых данных и подписку наблюдателей на эти уведомления. Он имеет простой интерфейс (всего два метода: `emit()` и `subscribe()`), позволяющий посылать пользовательские события и получать их как синхронно, так и асинхронно. Более подробно о наблюдении за данными рассказывается в *главе 6 «Асинхронная обработка данных в Angular 2»*. А сейчас ограничимся лишь констатацией, что программный интерфейс класса `EventEmitter` порождает события, которые могут обрабатываться методами компонентов, содержащих компонент, посылающий события. Эти события можно сделать видимыми за пределами компонента с помощью любого из его свойств, аннотированного декоратором `@Output()`.

Следующий фрагмент демонстрирует реализацию, вытекающую из предыдущего примера:

```
@Component({
  selector: 'countdown',
  template: '<h1>Time left: {{seconds}}</h1>'
})
```

```
class CountdownComponent {
  @Input() seconds: number;
  intervalId: number;
  @Output() complete: EventEmitter<any> = new EventEmitter();

  constructor() {
    this.intervalId = setInterval(() => this.tick(), 1000);
  }

  private tick(): void {
    if (--this.seconds < 1) {
      clearTimeout(this.intervalId);
      // Послать событие по окончании обратного отсчета
      this.complete.emit(null);
    }
  }
}
```

Новое свойство `complete` аннотируется типом `EventEmitter` и тут же инициализируется. Позже его метод `emit` будет использоваться для отправки события по окончании обратного отсчета. Методу `emit()` необходим один обязательный параметр типа `any` для отправки подписчикам данных о событии (или значения `null`, если они не нужны).

А теперь настроим принимающий компонент, чтобы он мог получать событие из выходного свойства `complete`, и реализуем обработчик:

```
@Component({
  selector: 'pomodoro-timer',
  directives: [CountdownComponent],
  template: `<div class="container text-center">
              <img src="assets/img/pomodoro.png" />
              <countdown [seconds]="25"
                (complete)="onCountdownCompleted()">
              </countdown>
            </div>`
})
class PomodoroTimerComponent {
  onCountdownCompleted(): void {
    alert('Time up!');
  }
}
```



Почему complete, а не onComplete?

Для свойств ввода/вывода Angular 2 поддерживает альтернативный синтаксис, называемый *канонической формой*. Например, на входное свойство, объявленное как `[seconds]`, можно сослать-

ся по имени `bind-seconds`, без квадратных скобок, а на выходное свойство, объявленное как `(complete)`, – по имени `on-complete`. Поэтому в именах выходных свойств никогда не используется префикс `on`, так как в противном случае придется смириться с именем `on-on-complete`, если будет решено использовать в проекте канонической синтаксис.

Передача данных в пользовательских событиях

Теперь, когда ясно, как посылать пользовательские события, почему бы не сделать еще один шаг и не отправить данные в охватывающий компонент? Мы уже знаем, что метод `emit()` класса `EventEmitter<T>` предусматривает в своей сигнатуре возможность передачи данных любого типа, определяемого с помощью аннотации `T`. Давайте добавим в пример отправку уведомлений о ходе обратного отсчета. Для чего это может понадобиться? По сути, компонент лишь выводит значение счетчика времени на экран, но было бы неплохо иметь возможность программно наблюдать за состоянием счетчика, чтобы выполнить какие-то действия, когда счетчик обнулится или достигнет определенного значения.

Добавим в компонент таймера еще одно выходное свойство и будем посылать событие после каждого изменения свойства `seconds`:

```
class CountdownComponent {
  @Input() seconds: number;
  intervalId: number;
  @Output() complete: EventEmitter<any> = new EventEmitter();
  @Output() progress: EventEmitter<number> = new EventEmitter();

  constructor() {
    this.intervalId = setInterval(() => this.tick(), 1000);
  }

  private tick(): void {
    if (--this.seconds < 1) {
      clearTimeout(this.intervalId);
      this.complete.emit(null);
    }
    this.progress.emit(this.seconds);
  }
}
```

Теперь переделаем шаблон принимающего компонента, чтобы он отображал оставшееся время. Эта функция уже имеется, но она реализована внутри компонента `CountdownComponent`. Теперь же реализуем вывод счетчика вне этого компонента:

```
@Component({
  selector: 'pomodoro-timer',
  directives: [CountdownComponent],
  template: `<div class="container text-center">
              <img src="assets/img/pomodoro.png" />
              <countdown [seconds]="25"
                (progress)="timeout = $event"
                (complete)="onCountdownCompleted()">
              </countdown>
              <p *ngIf="timeout < 10">
                Beware! Only
                <strong>{{timeout}} seconds</strong>
                left.
              </p>
            </div>`
})
class PomodoroTimerComponent {

  timeout: number;
  onCountdownCompleted(): void {
    alert('Time up!');
  }
}
```

Эти изменения позволили сделать значение `timeout` свойством принимающего компонента. А это дает возможность привязать обновление значений данного свойства к пользовательским обработчикам событий, как было сделано в предыдущем примере. Вместо привязки метода обработки событий к обработчику `(progress)` создана ссылка на зарезервированную переменную `$event`. Это указатель на выходное свойство `progress`, отражающее значение, переданное в функцию `emit()` при вызове `this.progress.emit(this.seconds)`. Короче говоря, `$event` является значением, которое принимает `this.seconds` внутри `CountdownComponent`. Присваивая значение свойству `timeout` внутри шаблона, мы также обновляем привязку в абзаце, добавленном в этот шаблон. Данный абзац становится видимым, когда `timeout` достигает значения меньше 10.

```
<countdown [seconds]="25"
  on-progress="timeout = $event"
  on-complete="onCountdownCompleted()">
</countdown>
```

Локальные ссылки в шаблонах

Выше мы видели, как привязывать данные к шаблонам с применением синтаксиса двойных фигурных скобок механизма интерполяции. Кроме того, в элементах, принадлежащих компонентам или даже обычным элементам управления HTML, нам достаточно часто придется использовать идентификаторы с предшествующим символом решетки (#). Такие идентификаторы, называемые локальными именами, используются для ссылок на компоненты в представлениях, чтобы в дальнейшем получить к ним доступ. Кроме того, они могут использоваться компонентами для ссылок на другие элементы виртуальной модели DOM и доступа к их свойствам.

В предыдущем разделе было показано, как организовать слежение за процессом обратного отсчета через событие `progress`. Но возможно ли полноценно использовать компонент или, по крайней мере, его общедоступные свойства и методы и читать значение свойства `seconds` при каждой его смене без использования события `progress`? Настройка ссылки на компонент открывает такую возможность.

Если пометить экземпляр компонента `CountdownComponent` в шаблоне `PomodoroTimerComponent` локальной ссылкой `#counter`, у нас появится возможность прямого доступа к общедоступным свойствам компонента, таким как `seconds`, и их привязки к другим элементам шаблона. Благодаря этому отпадает необходимость полагаться на событие `progress` или поле класса `timeout`, и мы сможем напрямую работать со значениями таких свойств, как демонстрирует следующий пример:

```
@Component({
  selector: 'pomodoro-timer',
  directives: [CountdownComponent],
  encapsulation: ViewEncapsulation.None,
  template: `<div class="container text-center">
              <img src="assets/img/pomodoro.png" />
<countdown  [seconds]="25"
  (complete)="onCountdownCompleted()"
  #counter>
</countdown>
<p>
  <button
    class="btn btn-default"
    (click)="counter.seconds = 25">
    Reset countdown to 25 seconds
  </button>
```

```
      </p>
      <p *ngIf="counter.seconds < 10">
        Beware! Only
        <strong>{{counter.seconds}} seconds</strong>
        left.
      </p>
    </div>`
})
class PomodoroTimerComponent {
  // timeout: number; /* Больше не требуется */
  onCountdownCompleted(): void {
    alert('Time up!');
  }
}
```

Альтернативный синтаксис для входных и выходных свойств

Помимо декораторов `@Input()` и `@Output()`, существует альтернативный синтаксис, позволяющий определять входные и выходные свойства компонентов с помощью декоратора `@Component`. Реализация метаданных в этом декораторе обеспечивает поддержку обеих возможностей посредством определения свойств `inputs` и `outputs` соответственно.

При таком подходе программный интерфейс компонента `CountdownComponent` можно реализовать следующим образом:

```
@Component({
  selector: 'countdown',
  template: '<h1>Time left: {{seconds}}</h1>',
  inputs: ['seconds'],
  outputs: ['complete', 'progress']
})
class CountdownComponent {
  seconds: number;
  intervalId: number;
  complete: EventEmitter<any> = new EventEmitter();
  progress: EventEmitter<number> = new EventEmitter();

  // Прочее...
}
```

Использовать такой синтаксис не рекомендуется, и он приведен здесь исключительно для иллюстрации. Во-первых, здесь дублируются определения имен точек входа, что увеличивает риск появления ошибок при модернизации кода. Это также противоречит соглаше-

нию о более экономной, насколько это возможно, реализации декораторов для улучшения читаемости.



Я настоятельно рекомендую использовать только декораторы `@Input` и `@Output`.

Настройка шаблона класса компонента

Декоратор `@Component` имеет также несколько дополнительных параметров, упрощающих управление шаблоном и его настройку. С другой стороны, фреймворк Angular 2 использует инкапсуляцию CSS для поддержки функциональных возможностей веб-компонентов.

Внутренние и внешние шаблоны

С увеличением размеров и сложности приложения есть вероятность, что будут разрастаться и шаблоны, включая в себя другие компоненты и большие куски разметки HTML. Встраивание всего этого кода в определения классов компонентов может превратиться в утомительную и неприятную задачу, при решении которой легко допустить ошибку. Чтобы избежать этого, можно использовать свойство `templateUrl`, указывающее на отдельный файл с разметкой HTML компонента.

Так, в предыдущем примере можно улучшить декоратор `@Component` класса `PomodoroTimerComponent`, заменив шаблон ссылкой на внешний HTML-файл. Создайте новый файл с именем `pomodoro-timer.html` в папке с файлом `pomodoro-timer.ts` и добавьте в него разметку HTML, которая сейчас находится в классе `PomodoroTimerComponent`:

```
<div class="container text-center">
  <img src="assets/img/pomodoro.png" alt="Pomodoro"/>
  <countdown
    [seconds]="25"
    (complete)="onCountdownCompleted()"
    #counter>
</countdown>
<p>
  <button
    class="btn btn-default"
    (click)="counter.seconds = 25">
    Reset countdown to 25 seconds
  </button>
</p>
```

```
  <p *ngIf="counter.seconds < 10">
    Beware! Only
    <strong>{{ counter.seconds }} seconds</strong>
    left.
  </p>
</div>
```

Теперь почистим декоратор `@Component`, заменив HTML-разметку в метаданных декоратора ссылкой на файл:

```
@Component({
  selector: 'pomodoro-timer',
  directives: [CountdownComponent],
  templateUrl: './pomodoro-tasks.html'
})
class PomodoroTimerComponent {
  // Определение класса...
}
```



Внешние шаблоны подчиняются в Angular 2 некоторым соглашениям, определяющим наиболее популярный стиль оформления кода в Angular 2 и требующим использовать имя файла для шаблона, совпадающее с именем соответствующего компонента, с добавлением любых префиксов или суффиксов. Более подробно соглашения об именовании рассматриваются в *главе 5 «Создание приложений с помощью компонентов Angular 2»*. Так будет легче догадаться или найти HTML-файл с шаблоном для конкретного компонента.

Когда следует переходить к созданию автономного шаблона вместо размещения разметки шаблона внутри компонента? Это зависит от сложности и размера шаблона. В этом случае лучшим советчиком будет здравый смысл.

Инкапсуляции CSS-стилей

Для более полной инкапсуляции кода и подготовки его к многократному использованию можно определить CSS-стили внутри компонентов. Такое размещение таблиц стилей позволяет сделать компоненты удобными в использовании и сопровождении. Имеются три способа определения CSS-стилей компонентов.

Свойства styles

Стили HTML-элементов и имена классов можно определить через свойство `styles` декоратора `@Component`:

```
@Component({
  selector: 'my-component',
  styles: [`
    p {
      text-align: center;
    }
    table {
      margin: auto;
    }
  `]
})
```

Это свойство принимает массив строк с правилами CSS. Они применяются к разметке шаблона путем их внедрения в заголовок документа на этапе инициализации приложения. Правила можно разместить в одну строку или воспользоваться синтаксисом шаблонных строк из спецификации ES2015, чтобы оформить код с отступами и сделать его более читаемым, как показано в примере выше.

Свойство styleUrls

Подобно свойству `styles`, свойство `styleUrls` принимает массив строк, с той лишь разницей, что каждая строка является ссылкой на внешнюю таблицу стилей. Это свойство может использоваться вместе со свойством `styles`, определяя при необходимости другие наборы правил:

```
@Component({
  selector: 'my-component',
  styleUrls: ['path/to/my-stylesheet.css'],
  styles: [`
    p {
      text-align: center;
    }
    table {
      margin: auto;
    }
  `]
})
```

Встроенные таблицы стилей

Стили можно также встраивать непосредственно в сам шаблон, независимо от того, является ли он встроенным или передается через параметр `templateUrl`:

```
@Component({
  selector: 'app',
  template: `
    <style> p { color: red; } </style>
    <p>I am a red paragraph</p>`
})
```

Управление инкапсуляцией представлений

Все предыдущие способы (`styles`, `styleUrls` и встроенные таблицы стилей) подчиняются обычным правилам специфичности CSS (https://developer.mozilla.org/en/docs/Web/CSS/Specificity). Управление специфичностью стилей CSS упрощается в браузерах с поддержкой теневой модели DOM благодаря ограничению их области видимости. CSS-стили применяются к элементам, содержащимся в компоненте, но их действие не распространяется за его границы.

Однако Angular вставляет эти таблицы стилей в заголовок документа, поэтому они могут влиять и на другие элементы приложения. Чтобы предотвратить это, следует использовать различные уровни инкапсуляции представления.

По своей сути, инкапсуляция – это прием управления стилями CSS внутри компонентов, реализованный в Angular, который можно использовать как в браузерах с поддержкой теневой модели DOM, так и без нее. Управление осуществляется с помощью перечисления `ViewEncapsulation`, которое содержит следующие значения:

- **Emulated**: значение по умолчанию. Предназначено для эмуляции встроенной области видимости теневой модели DOM через ограничение области видимости CSS-правил конкретным селектором, указывающим на компонент. Этот вариант помогает оградить стили компонента от влияния прочих библиотек;
- **Native**: используется встроенный механизм инкапсуляции теневой модели DOM, который работает только в браузерах, поддерживающих ее;
- **None**: инкапсуляция шаблонов или стилей отсутствует. Стили будут помещаться в заголовок документа.

Проиллюстрируем это примером. Импортируем перечисление `ViewEncapsulation` в сценарий и создадим свойство `encapsulation` со значением `Emulated`. Затем определим правило оформления текста со значением счетчика, чтобы любой тег `<h1>` (!) окрашивался в темно-красный цвет:

```
import {
  Component,
  EventEmitter,
  Input, Output,
  ViewEncapsulation
} from '@angular/core';
import { bootstrap } from '@angular/platform-browser-dynamic';

@Component({
  selector: 'countdown',
  template: '<h1>Time left: {{seconds}}</h1>',
  styles: ['h1 { color: #900 }'],
  encapsulation: ViewEncapsulation.Emulated
})
class CountdownComponent {
  // Прочее...
}
```

А теперь запустите инспектора из набора инструментов разработчика в браузере и посмотрите, как Angular 2 внедрил правила CSS в сгенерированную им разметку HTML, в блок `<HEAD>`. Таблица стилей оказалась изолирована, чтобы обеспечить применение глобального CSS-правила весьма конкретным способом ко всем `<h1>` элементам, исключительно в рамках компонента `CountdownComponent`.

Рекомендуем самостоятельно опробовать разные значения инкапсуляции и посмотреть, как будет помещаться CSS-код в документ. Вы сразу заметите разные виды изоляции, которые обеспечивает каждый из вариантов.

Итоги

В этой главе были рассмотрены доступные в Angular 2 варианты создания мощного программного интерфейса компонентов, позволяющие обеспечить взаимодействия между компонентами, настройку их свойств с помощью статических значений или управляемых привязок. Также было описано, как заключить один компонент в другой, как создавать экземпляры пользовательских элементов в своем шаблоне и как конструировать большие деревья компонентов в приложениях. Выходные параметры реализуют интерактивный слой, превращая компоненты в источники событий, позволяя им успешно общаться с любыми родительскими компонентами, способными размещать их. Ссылки в шаблонах дают возможность создавать ссылки на пользова-

тельские элементы и использовать их для доступа к свойствам и методам этих элементов из шаблона декларативным образом. Также был описан процесс выделения HTML-шаблонов компонентов во внешние файлы, чтобы облегчить их дальнейшее сопровождение. То же относится и к таблицам стилей, которые требуется привязать к компоненту, если решено не использовать стили, встроенные в компонент. В обзоре поддержки инкапсуляции представлений в Angular 2 мы познакомились с несколькими способами использования областей видимости для стилей CSS-компонента в теневой модели DOM и возможностью ее эмуляции, когда она не поддерживается.

Нам предстоит узнать еще больше об управлении шаблонами в Angular 2, основанном главным образом на двух концепциях, широко применяемых в Angular. Я имею в виду директивы и фильтры, которые подробно рассматриваются в *главе 4* «*Расширение компонентов с помощью директив и фильтров*».

Глава 4

Расширение компонентов с помощью директив и фильтров

В предыдущих главах мы создали несколько компонентов, отображающих данные на экране с помощью входных и выходных свойств. В этой главе мы применим новые знания, чтобы поднять компоненты на следующий уровень, использовав директивы и фильтры. Фильтры, если говорить кратко, дают возможность упорядочивать и преобразовывать информацию, включаемую в шаблоны, директивы же предоставляют более масштабные возможности, позволяющие получить доступ к свойствам вмещающего элемента, а также связать пользовательские обработчики событий с данными.

В этой главе мы:

- познакомимся с набором встроенных директив фреймворка Angular 2;
- узнаем, как с помощью фильтров организовать фильтрацию выводимых данных;
- спроектируем и реализуем собственные фильтры и директивы;
- используем встроенные объекты для работы с шаблонами;
- применим полученные знания на практике, продолжив работу над проектом Pomodoro, и разработаем полноценную интерактивную таблицу со списком задач.

Директивы в Angular 2

Фреймворк Angular 2 определяет директивы как компоненты без представлений. Фактически компоненты являются директивами, связанными с представлениями. Такое различие делается из-за того, что директивы являются неотъемлемой частью ядра Angular 2, и все они (обычные директивы и директивы-компоненты) не могут существовать друг без друга. Директивы могут влиять на поведение и способ отображения содержимого HTML-элементов или пользовательских элементов.

Встроенные директивы

Познакомимся поближе с директивами ядра фреймворка, а затем в этой же главе займемся разработкой собственной директивы.

Директива NgIf

Как утверждает официальная документация, директива `NgIf` удаляет или воссоздает часть дерева модели DOM на основании выражения. Если результатом выражения, присвоенного директиве `NgIf`, является значение `false`, элемент удаляется из модели DOM. В противном случае создается клон элемента. С помощью этой директивы можно усовершенствовать таймер обратного отсчета, как показано ниже:

```
<pomodoro-timer [seconds]="timeout"></pomodoro-timer>
<p *ngIf="timeout === 0">Time up!</p>
```

Когда значение таймера достигнет `0`, на экране появится абзац с текстом `Time up!`. Вы, вероятно, обратили внимание на звездочку перед именем директивы. Она указывает, что Angular должен встроить элемент HTML с директивой `NgIf` (и все его дочерние элементы, если имеются) в тег `<template>`, который в дальнейшем будет использоваться для отображения на экране. Описание особенностей интерпретации шаблонов в Angular выходит за рамки этой книги, поэтому просто заметим, что удобные синтаксические конструкции, поддерживаемые в Angular, являются сокращениями от других, более громоздких синтаксических конструкций, основывающихся на тегах `<template>`.

Возможно, вам интересно узнать, в чем разница между отображением на экране фрагментов HTML с помощью `*ngIf="условие"` и `[hidden]="условие"`. В первом случае фрагменты HTML встраи-

ваются в документ, когда условие принимает значение `true`, и удаляются, когда условие принимает значение `false`. Во втором случае фрагменты не встраиваются и не удаляются, а просто определяется их видимость.

Директива NgFor

Директива `NgFor` позволяет выполнить обход коллекции (или любого другого итерируемого объекта) и связать каждый из ее элементов с выбранным шаблоном, где имеется возможность удобно определить заполнители для размещения данных элемента. Каждый экземпляр шаблона ограничивается внешним контекстом, в котором находится директива цикла, благодаря чему имеется доступ к другим привязкам. Допустим, имеется компонент штатного расписания организации `Staff`, содержащий поле `employees` с массивом объектов сотрудников `Employee`. С помощью директивы `NgFor` можно перечислить имена сотрудников из поля `employees` и их должности, как показано ниже:

```
<ul>
  <li *ngFor="let employee of employees; let i = index; let last = last">
  Employee #{{i}}: - {{employee.name}}, {{employee.position}}
  <span *ngIf="last"><br />End of list</span>
  </li>
</ul>
```

Как показывает пример, на каждом шаге цикла происходит извлечение элемента из итерируемого объекта (коллекции) в локальную ссылку, что позволяет легко привязать этот элемент к шаблону. Здесь также используется удобная синтаксическая конструкция, которая уже применялась в предыдущем разделе. Кроме того, Angular позволяет присвоить локальной переменной порядковый номер текущей итерации `index` для передачи в контекст шаблона. Имеется также возможность присвоить локальной переменной значение `last`, чтобы с ее помощью сообщить о выводе последнего элемента списка.

Эта директива отслеживает изменения итерируемого объекта и добавляет, удаляет или упорядочивает отображаемые шаблоны при добавлении, удалении или сортировке элементов в коллекции.

Директива NgStyle

Как вы, наверное, уже догадались, эта директива позволяет подключать CSS-стили путем определения пользовательского объекта или выражения. С ее помощью можно подключить объект, ключи

и значения, определяющие CSS-свойства, или просто определить конкретные свойства и привязать к ним данные:

```
<p [ngStyle]="{ 'color': myColor, 'font-weight': myFontWeight }">
  I am red and bold
</p>
```

Если компонент определит свойства `myColor` и `myFontWeight` со значениями `red` и `bold` соответственно, текст получит указанные цвет и насыщеность шрифта. Директива всегда отражает происходящие в компоненте изменения и позволяет использовать объект вместо прямой привязки данных:

```
<p [ngStyle]="myCssConfig">I am red and bold</p>
```

Директива NgClass

Подобно директиве `NgStyle`, директива `NgClass` позволяет программно определять и изменять имена классов в элементах DOM с помощью удобного декларативного синтаксиса. Но этот синтаксис не лишен своих сложностей. Рассмотрим три случая для следующего примера:

```
<p [ng-class]="{{myClassNames}}">Hello Angular!</p>
```

Можно воспользоваться строковым типом, тогда если `myClassNames` содержит строку с одним или несколькими классами, разделенными пробелом, все они будут применены к параграфу.

Можно также использовать массив, что приведет к добавлению классов, содержащихся во всех его элементах.

И последнее, но важное замечание: можно использовать объект, в котором каждый ключ представляет имя CSS-класса и ссылается на логическое значение. Если ключу соответствует значение `true`, класс добавляется. В противном случае – удаляется. Обычно отдается предпочтение именно этому способу работы с именами классов.

Директивы NgSwitch, NgSwitchWhen и NgSwitchDefault

Директива `NgSwitch` используется для выбора отображаемого шаблона из предопределенного набора, в зависимости от условия. Ее реализация состоит из нескольких частей, поэтому в этот раздел включено описание сразу трех различных директив.

Директива `NgSwitch` вычисляет заданное выражение и включает или выключает отображение дочерних элементов с атрибутами `ngSwitchWhen`, значения которых совпадают с результатом выражения

в родительском элементе ngSwitch. Особого упоминания требует дочерний элемент с атрибутом ngSwitchDefault. Этот атрибут определяет шаблон, который отображается, когда нет ни одного элемента, в котором значение атрибута ngSwitchWhen совпадало бы с результатом выражения в родительской директиве.

Все это иллюстрирует следующий пример:

```
<div [ngSwitch]="weatherForecastDay">
  <template ngSwitchWhen="today">{{weatherToday}}</template>
  <template ngSwitchWhen="tomorrow">
    {{weatherTomorrow}}</template>
  <template ngSwitchDefault>
    Pick a day to see the weather forecast
  </template>
</div>
```

Родительская директива [ngSwitch] определяет значение контекстной переменной weatherForecastDay, и каждая вложенная директива ngSwitchWhen сравнивается с полученным значением. Вместо переменной можно использовать выражение, но тогда ngSwitchWhen следует заключить в скобки, чтобы Angular интерпретировал их содержимое как контекстные переменные, а не как текстовые строки.



Когда работа над этой книгой подходила к концу, команда Angular обсуждала возможность переименования директивы ngSwitchWhen в ngSwitchCase, чтобы привести ее в соответствие с ключевыми словами switch/case в языках JavaScript и TypeScript, а также с другими директивами i18n, такими как NgPlural и NgPluralCase. Вполне вероятно, что это критическое изменение будет сделано в заключительной версии Angular 2, поэтому уточните окончательный синтаксис директивы NgSwitch в электронной документации.
Описание директив NgPlural и NgPluralCase не включено в книгу. Они предоставляют удобный способ вывода или удаления шаблонов блоков модели DOM, соответствующих переключающему выражению, возвращающему число или строку, как в директивах NgSwitch и NgSwitchWhen.

Управление привязками шаблонов с помощью фильтров

Теперь мы знаем, как с помощью директив организовать отображение содержимого в зависимости от данных, которыми управляют классы компонентов, но существует еще одна мощная возможность, которая

часто используется при разработке с применением Angular. Имеются в виду *фильтры* (*pipes*).

Фильтры позволяют на уровне представлений фильтровать и передавать результаты выражений для преобразования или простого форматирования отображаемых данных. Их синтаксис довольно прост и состоит из имени фильтра, следующего за преобразуемым выражением, отделенного от него символом вертикальной черты:

```
@Component({
  selector: "greeting",
  template: "HELLO {{ name | uppercase }}"
})
class GreetingComponent {
  name: string;
}
```

В предыдущем примере приветствие всегда выводится на экран прописными буквами. Поскольку заранее неизвестно, будет ли имя написано буквами в верхнем регистре, такой способ гарантирует вывод в требуемом формате путем преобразования букв в имени в верхний регистр. Фильтры могут объединяться в цепочки, и фреймворк Angular содержит широкий ассортимент встроенных фильтров. В этой главе будет также описан процесс создания собственных фильтров для преобразования данных перед выводом, когда встроенных фильтров для этого недостаточно.

Фильтры uppercase/lowercase

Имена фильтров `uppercase`/`lowercase` ясно определяют их назначение. Как показано в примере выше, эти фильтры преобразуют строковое значение в верхний или нижний регистр. Вставьте следующий код в любое место в представлении и проверьте вывод:

```
<p>{{ 'hello world' | uppercase }}</p>
<!-- вывод: 'HELLO WORLD' -->

<p>{{ 'wEIrD hElLo' | lowercase }}</p>
<!-- вывод: 'weird hello' -->
```

Фильтры number, percent и currency

Числовые данные могут выводиться самыми разными способами, и такие фильтры очень удобны для форматирования и локализации вывода. Эти фильтры используют Internationalization API, и,

следовательно, их правильную работу гарантируют лишь браузеры Chrome и Opera.

Фильтр number

Фильтр `number` помогает определить группировку и размеры чисел с учетом региональных настроек в браузере. Он имеют следующий синтаксис:

```
expression | number[:digitInfo]
```

Здесь `expression` представляет число, а `digitInfo` имеет следующий формат:

```
{minIntegerDigits}.{minFractionDigits}-{maxFractionDigits}
```

Эти привязки означают следующее:

- `minIntegerDigits`: минимальное количество цифр в целой части числа. По умолчанию 1;
- `minFractionDigits`: минимальное количество цифр в дробной части числа. По умолчанию 0;
- `maxFractionDigits`: максимальное количество цифр в дробной части числа. По умолчанию 3.



Имейте в виду, что эти значения и другие детали зависят от фактической реализации интернационализации.

Фильтр percent

Фильтр `percent` форматирует числа в соответствии с локальным форматом процентов. Помимо этого, он наследует от фильтра `number` возможность определения размеров целых и десятичных частей и имеет следующий формат:

```
expression | percent[:digitInfo]
```

Фильтр currency

Форматирует числа как значения сумм в валюте, позволяя указывать код валюты, например `USD` для доллара США и `EUR` для евро, в соответствии с настройками отображения. Этот фильтр имеет следующий формат:

```
expression | currency[:currencyCode[:symbolDisplay[:digitInfo]]]
```

В этой инструкции `currencyCode` обозначает код валюты согласно стандарту ISO 4217, а логический флаг `symbolDisplay` указывает, что выводить: символ валюты (например, `$`) или ее код (например, `USD`).

По умолчанию флаг имеет значение `false`. Подобно фильтрам `number` и `percent`, здесь также можно форматировать результат, определяя размеры целой и дробной частей с помощью `digitInfo`:

```
<p>{{ 11256.569 | currency:"GBP":true:'4.1-2' }}</p>
<!-- вывод: '£11,256.57' -->
```

Фильтр slice

Этот фильтр действует эквивалентно методам `Array.prototype.slice()` и `String.prototype.slice()` и служит для выделения подмножеств (`slice`) из коллекций: списков, массивов или строк. Его синтаксис достаточно прост и подчиняется тем же соглашениям, что и упомянутые выше методы `slice()`:

```
expression | slice:start[:end]
```

Фильтр требует задать начальный индекс, с которого начнется извлечение элементов массива или символов строки, и позволяет указать необязательный конечный индекс, значением которого по умолчанию становится индекс последнего элемента входной коллекции.



В аргументах `start` и `end` допускается передавать как положительные, так и отрицательные значения, так же как в методе `slice()`. Более подробно обо всех возможностях можно узнать из документации с описанием JavaScript API.

И последнее, но важное замечание: обратите внимание, что при работе с коллекциями возвращаемый список всегда является копией, даже когда возвращаются все элементы.

Фильтр date

Как нетрудно догадаться, фильтр `date` преобразует значения дат в строку в соответствии с заданным форматом. В качестве часового пояса форматированной даты будет выбран часовой пояс локальной системы конечного пользователя. Фильтр имеет очень простой синтаксис:

```
expression | date[:format]
```

Выражение `expression` должно быть объектом `Date` или числом (с количеством миллисекунд, прошедших с начала эпохи UTC). Аргумент формата `format` поддерживает широкий диапазон вариантов, базирующихся на символах даты и времени. Для большего удобства имеется несколько коротких псевдонимов, представляющих наиболее распространенные форматы:

- 'medium': эквивалент формата 'yMMMdjms' (например, «Sep 3, 2010, 12:05:08 PM» для локали en-US);
- 'short': эквивалент формата 'yMdjm' (например, «9/3/2010, 12:05 PM» для локали en-US);
- 'fullDate': эквивалент формата 'yMMMMEEEEd' (например, «Friday, September 3, 2010» для локали en-US);
- 'longDate': эквивалент формата 'yMMMMd' (например, «September 3, 2010»);
- 'mediumDate': эквивалент формата 'yMMMd' (например, «Sep 3, 2010» для локали en-US);
- 'shortDate': эквивалент формата 'yMd' (например, «9/3/2010» для локали en-US);
- 'mediumTime': эквивалент формата 'jms' (например, «12:05:08 PM» для локали en-US);
- 'shortTime': эквивалент формата 'jm' (например, «12:05 PM» для локали en-US).

Фильтр JSON

Фильтр JSON имеет, пожалуй, самое простое определение, он принимает объект и преобразует его в формате JSON:

```
{{ { name: 'Eve', age: 43 } | json }}
```

Результат:

```
{ "name": "Eve", "age": 43 }
```

Фильтр replace

Фильтр replace во многом схож с функцией String.prototype.replace() и предназначен для обработки строковых выражений и чисел, которые интерпретируются как строки, в соответствии с заданным шаблоном. Все совпадения будут заменены заданной строкой замены. Вместо строки замены можно указать функцию или ссылку на функцию, которой будут передаваться найденные соответствия. В общем виде синтаксис выглядит следующим образом:

```
expression | replace:pattern:replacement
```

Важно отметить, что шаблон может быть регулярным выражением. С учетом того, что для поиска соответствий Angular 2 внутренне использует регулярные выражения, экранируйте специальные символы, такие как круглые и квадратные скобки, и т. д.

Фильтры интернационализации

Поскольку фреймворк Angular строго придерживается политики поддержки интернационализации, он предоставляет набор фильтров, делающих доступным использование возможностей i18n. В этой книге будут описаны только два таких фильтра, но вполне вероятно, что в будущем будут добавлены и другие. За дополнительной информацией обращайтесь к официальной документации.

Фильтр i18nPlural

Фильтр `i18nPlural` очень прост: он проверяет числовое значение и использует объект, отображающий число в соответствующую строку. Таким способом можно организовать вывод разных строк для числа, равного нулю, единице, двум, больше N и т. д. В проекте Pomodoro, к примеру, можно было бы использовать шаблон:

```
<h1> {{ pomodoros | i18nPlural:pomodorosWarningMapping }} </h1>
```

предварительно определив поле контроллера компонента с отображением:

```
class MyPomodorosComponent {
  pomodoros: number;
  pomodorosWarningMapping: any = {
    '=0': 'No pomodoros for today',
    '=1': 'One pomodoro pending',
    'other': '# pomodoros pending'
  }
}
```

Имеется даже возможность вставить оцениваемое числовое значение в выводимую строку с помощью заполнителя #. Если совпадающее значение не найдено, фильтр вернет строку, соответствующую ключу `'other'`.

Фильтр i18nSelect

Фильтр `i18nSelect` похож на фильтр `i18nPlural`, но, в отличие от него, анализирует строковое значение. Этот фильтр идеально подходит для вставки локализованного текста или меток в зависимости от изменения состояния. Например, в таймер Pomodoro можно добавить поддержку пользовательского интерфейса на нескольких языках:

```
<button (click)="togglePause()">
  {{ languageCode | i18nSelect: localizedLabelsMap }}
</button>
```

Поле localizedLabelsMap в классе контроллера можно определить так:

```
class PomodoroTimerComponent {
  languageCode: string = 'fr';
  localizedLabelsMap: any = {
    'en': 'Start timer',
    'es': 'Comenzar temporizador',
    'fr': 'Démarrer une séquence',
    'other': 'Start timer'
  }
  ...
}
```

Важно отметить, что этот удобный фильтр можно использовать не только для локализации компонентов, но и для получения строк, соответствующих ключам. Если фильтр i18nSelect не найдет явно указанного ключа, он вернет строку, соответствующую ключу 'other'.

Фильтр async

Иногда требуется наблюдать за данными, которые обрабатываются асинхронно классом компонента, и обеспечивать оперативное обновление отображаемой информации по изменению наблюдаемых полей или завершению асинхронной загрузки, происходящих после вывода представления. Фильтр async подписывается на уведомления об изменении значения или о завершении асинхронной операции и возвращает последнее полученное значение. При получении нового значения фильтр async отмечает компонент как подлежащий проверке на наличие изменений.

Реализация списка заданий в проекте Pomodoro

Теперь, после знакомства со всеми элементами, позволяющими конструировать полноценные компоненты, пришло время применить вновь приобретенные знания на практике. На нескольких следующих страницах описывается процесс реализации диспетчера списка заданий для приложения Pomodoro. На рис. 4.1 изображена таблица со списком заданий, которую мы должны получить в результате:

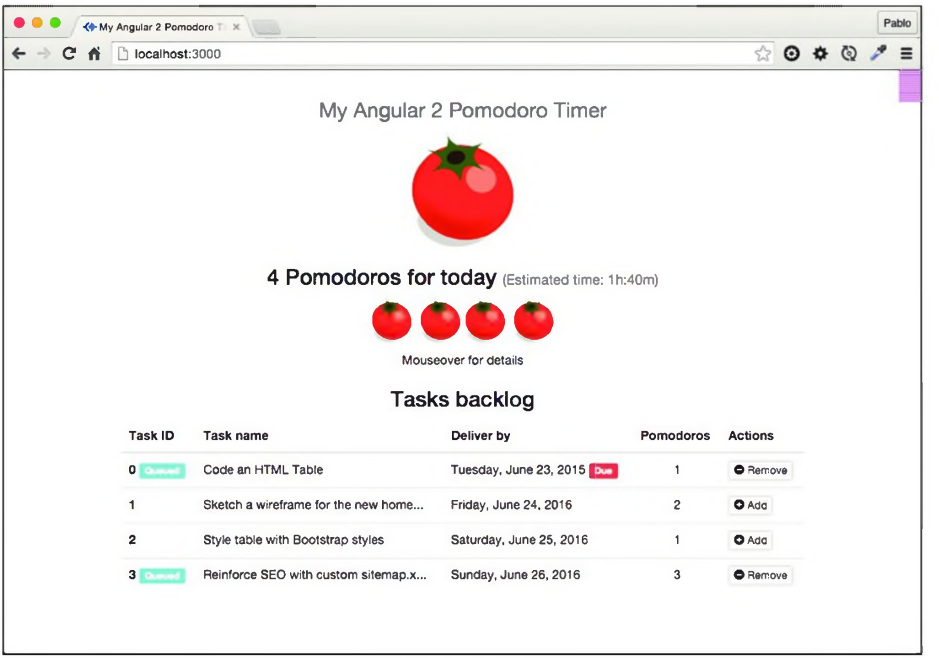


Рис. 4.1

Мы также создадим очередь из доступных, но невыполненных заданий. Это поможет показать время, необходимое для выполнения всех заданий в очереди, и вывести количество Pomodoro-сеансов, запланированных на текущий день.

Создание основного HTML-контейнера

Перед конструированием компонента необходимо подготовить рабочее окружение, для чего повторно используем тот же стандартный HTML-файл, что использовался при работе с предыдущим компонентом. Чтобы сохранить проделанную ранее работу, скопируйте в отдельное место файлы `package.json`, `tsconfig.json`, `typings.json` и `index.html`, созданные в процессе работы с предыдущими примерами. Повторно установите необходимые модули, если потребуется, и замените содержимое тега `body` в шаблоне `index.html`, как показано ниже:

```
<nav class="navbar navbar-default navbar-static-top">
  <div class="container">
    <div class="navbar-header">
      <strong class="navbar-brand">My Pomodoro Tasks</strong>
    </div>
```

```
  </div>
</nav>
<pomodoro-tasks></pomodoro-tasks>
```

По сути, мы лишь добавили в заголовок макета новый пользовательский элемент `<pomodoro-tasks>`, заменивший предыдущий `<pomodoro-timer>`. Далее следует изменить команду `System.import()`, указав в ней новый скомпилированный класс компонента:

```
System.import('built/pomodoro-tasks')
      .then(null, console.error.bind(console));
```

Построение таблицы со списком заданий с помощью директив Angular

Создайте пустой файл `pomodoro-tasks.ts`. Вновь созданный файл предназначен для построения нового компонента и включения в него определений всех фильтров, директив и компонентов, которые будут рассматриваться в этой главе.



На практике проекты никогда не реализуются подобным образом, поскольку код должен соответствовать принципу «один класс – один файл» и использовать ECMAScript-модули для сборки воедино. В *главе 5 «Создание приложений с помощью компонентов Angular 2»* будет приведен общий обзор передового опыта создания приложений с Angular 2, включая стратегию правильной организации дерева каталогов и элементов (компонентов, директив, фильтров, служб и т. д.). В этой главе, напротив, весь код будет содержаться в одном файле `pomodoro-tasks.ts`, что позволит обсуждать все рассматриваемые темы, без необходимости переключаться между файлами. Имейте в виду, что такой подход противоречит общепринятому и применен здесь исключительно в учебных целях. Порядок объявления элементов в файле весьма важен. И при необходимости обращайтесь к репозиторию на GitHub.

Прежде чем начать работу с компонентом, импортируем требуемые зависимости, формализуем модель данных, используемую для заполнения таблицы, и подготовим некоторые данные, которые будут обрабатываться служебным классом.

Начнем с добавления в файл `pomodoro-tasks.ts` следующего блока кода и импорта всего необходимого, что потребуется в этой главе. Обратите особое внимание на имена, импортируемые из библиотеки Angular 2. С декораторами `Component` и `Input` мы уже знакомы, а все прочие будут описаны далее в этой главе:

```
import {
  Component,
  Input,
  Pipe,
  PipeTransform,
  Directive,
  OnInit,
  HostListener
} from '@angular/core';
import { bootstrap } from '@angular/platform-browser-dynamic';
```

Сразу вслед за инструкциями импорта определим модель данных для заданий:

```
/// Интерфейс модели
interface Task {
  name: string;
  deadline: Date;
  queued: boolean;
  pomodorosRequired: number;
}
```

Схема интерфейса модели `Task` достаточно очевидна. Каждое задание имеет название, срок выполнения, количество Pomodoro-сеансов для его выполнения и признак `queued` нахождения задания в очереди.



Возможно, вас удивило, что модель определена с помощью интерфейса, а не класса, однако это наиболее удачное решение, когда модель не обладает бизнес-логикой, требующей реализации методов, преобразования данных в конструкторе или функций записи и чтения значений свойств. Когда ничего этого не требуется, достаточно интерфейса, поскольку он обеспечивает статическую типизацию, большую простоту и наглядность.

Далее определим некоторые данные и класс-оболочку службы, возвращающей их в виде коллекции объектов `Task`. Определенный здесь класс `TaskService` использует хитрый прием, поэтому добавим его сразу после интерфейса `Task`:

```
/// Локальная служба
class TaskService {
  public taskStore: Array<Task> = [];

  constructor() {
    const tasks = [
      {
        name: "Code an HTML Table",
```

```
    deadline: "Jun 23 2015",
    pomodorosRequired: 1
  }, {
    name: "Sketch a wireframe for the new homepage",
    deadline: "Jun 24 2016",
    pomodorosRequired: 2
  }, {
    name: "Style table with Bootstrap styles",
    deadline: "Jun 25 2016",
    pomodorosRequired: 1
  }, {
    name: "Reinforce SEO with custom sitemap.xml",
    deadline: "Jun 26 2016",
    pomodorosRequired: 3
  }
];

this.taskStore = tasks.map(task => {
  return {
    name: task.name,
    deadline: new Date(task.deadline),
    queued: false,
    pomodorosRequired: task.pomodorosRequired
  };
});
  }
}
```

Этот класс хранилища данных едва ли требует дополнительных пояснений: он экспортирует свойство `taskStore`, которое возвращает массив объектов, реализующих интерфейс `Task` (со всеми выгодами статической типизации), предоставляющий информацию об имени, сроке и времени исполнения, измеряемом в Pomodoro-сеансах.

Теперь, когда у нас имеется хранилище данных и класс модели, можно начинать разработку компонента Angular, который будет использовать это хранилище для отображения заданий в шаблоне представления. Добавьте следующую реализацию компонента после уже написанного кода:

```
/// Классы компонентов

/// - Главный родительский компонент

@Component({
  selector: 'pomodoro-tasks',
  styleUrls: ['pomodoro-tasks.css'],
```

```
  templateUrl: 'pomodoro-tasks.html'
})
class TasksComponent {
  today: Date;
  tasks: Task[];

  constructor() {
    const TasksService: TasksService = new TasksService();
    this.tasks = taskService.taskStore;
    this.today = new Date();
  }
};

bootstrap(TasksComponent);
```

Здесь определяется новый компонент `TasksComponent` с селектором `<pomodoro-tasks>` (он уже был внедрен в главный файл `index.html`, помните?), и посредством функции `bootstrap` создается его экземпляр. Этот класс экспортирует два свойства: текущую дату и коллекцию заданий, которые будут отображены в таблице, содержащейся в представлении компонента, как будет показано чуть ниже. Для этого в конструкторе класса создается экземпляр источника данных, который был разработан ранее, и отображается в массив моделей типа `Task`, на который ссылается поле `tasks`. Также выполняется инициализация свойства `today` экземпляром встроенного объекта `Date`, который содержит текущую дату.



Как видите, имя селектора компонента не соответствует имени класса контроллера. Более подробно соглашения об именовании рассматриваются в конце этой главы, в качестве подготовки к *главе 5 «Создание приложений с помощью компонентов Angular 2»*.

А теперь создадим файл стилей с простым и понятным содержимым. Создайте новый файл `pomodoro-tasks.css` в том же каталоге, где размещается файл компонента, и поместите в него следующий набор правил:

```
h3, p {
  text-align: center;
}
table {
  margin: auto;
  max-width: 760px;
}
```

Вновь созданная таблица стилей настолько проста, что помещение ее в отдельный файл может показаться излишним. Однако это дает хорошую возможность продемонстрировать в примере применение свойства `styleUrls` метаданных компонента.

По-другому будет реализован и HTML-шаблон. На этот раз мы поместим HTML-шаблон не в компонент, а во внешний HTML-файл, что сделает код представления более управляемым. Создайте HTML-файл с именем `pomodoro-tasks.html` в том же каталоге, где располагается файл основного класса контроллера компонента, и заполните его следующей разметкой HTML:

```
<div class="container text-center">
  <img src="assets/img/pomodoro.png" alt="Pomodoro"/>
  <div class="container">
    <h4>Tasks backlog</h4>
    <table class="table">
      <thead>
        <tr>
          <th>Task ID</th>
          <th>Task name</th>
          <th>Deliver by</th>
          <th>Pomodoros</th>
          <th>Actions</th>
        </tr>
      </thead>
      <tbody>
        <tr *ngFor="let task of tasks; let i = index">
          <th scope="row">{{i}}</th>
          <td>{{task.name | slice: 0:35 }}
            <span [hidden]="task.name.length < 35">...</span>
          </td>
          <td>{{task.deadline | date: 'fullDate' }}
            <span *ngIf="task.deadline < today" class="label label-danger">
            Due
            </span>
          </td>
          <td class="text-center">{{task.pomodorosRequired}}</td>
          <td>
            [Future options...]
          </td>
        </tr>
      </tbody>
    </table>
  </div>
</div>
```

Мы создали таблицу, красиво оформленную с помощью фреймворка Bootstrap. Затем отобразили все задания с помощью директивы `NgFor`, извлекающей и выводящей порядковый номер `index` каждого элемента коллекции, как было описано при знакомстве с директивой `NgFor` выше в этой главе.

Обратите внимание, как форматируются названия заданий и сроки их выполнения с помощью фильтров и как названия завершаются многоточием, когда их длина превышает выделенное для них максимальное количество символов, что реализуется через HTML-свойство `hidden`, связанное с выражением Angular. Кроме того, когда срок выполнения задания оказывается в прошлом, добавляется красная этикетка, указывающая, что данное задание просрочено. Если выполнить приведенный выше код, он выведет страницу, показанную на рис. 4.2.

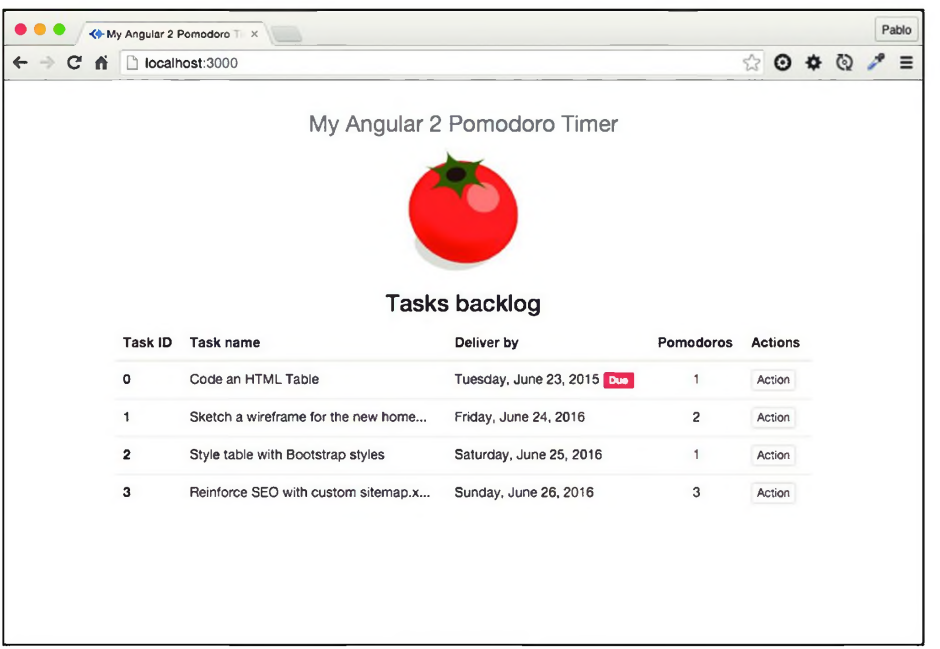


Рис. 4.2

Возможно, вы заметили, что кнопки **Action** в текущей реализации отсутствуют. Это будет исправлено в следующем разделе, где речь пойдет о состоянии компонентов. В *главе 1 «Создание самого первого компонента в Angular 2»* рассказывалось вкратце, как реализовать об-

работчик события щелчка, приостанавливающий и возобновляющий обратный отсчет Pomodoro-таймера, а в *главе 3 «Реализация свойств и событий компонентов»* мы более предметно обсудили приемы применения выходных свойств. Теперь продолжим исследования, заняв-шись реализацией подключения обработчиков событий модели DOM к общедоступным методам компонента для придания ему полноценной интерактивности.

Переключение заданий в списке

Добавьте следующий метод в класс контроллера `TasksComponent`. Он просто меняет значение свойства `queued` в указанном экземпляре объекта `Task`:

```
toggleTask(task: Task): void {
  task.queued = !task.queued;
}
```

Теперь осталось подключить этот метод к кнопкам в представлении. Внесем изменения в представление, добавив атрибут `click` (заключенный в скобки, чтобы он интерпретировался как выходное свойство) в кнопку, созданную с помощью директивы цикла `NgFor`. Теперь, когда объекты `Task` могут характеризоваться различными состояниями, отразим это на подписях кнопок с помощью структуры `NgSwitch`:

```
<table class="table">
  <thead>
    <tr>
      <th>Task ID</th>
      <th>Task name</th>
      <th>Deliver by</th>
      <th>Pomodoros</th>
      <th>Actions</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="#task of tasks; #i = index">
      <th scope="row">{{i}}
        <span *ngIf="task.queued" class="label label-info">
          Queued
        </span>
      </th>
      <td>{{task.name | slice: 0:35 }}
```

```
    <span [hidden]="task.name.length < 35">...</span>
  </td>
  <td>{{task.deadline | date: 'fullDate' }}
    <span *ngIf="task.deadline < today"
          class="label label-danger">
      Due
    </span>
  </td>
  <td class="text-center">{{task.pomodorosRequired}}</td>
  <td>
    <button
    type="button"
    class="btn btn-default btn-xs"
    (click)="toggleTask(task)"
    [ngSwitch]="task.queued">
      <template [ngSwitchWhen]="false">
        <i class="glyphicon glyphicon-plus-sign"></i>
        Add
      </template>
      <template [ngSwitchWhen]="true">
        <i class="glyphicon glyphicon-minus-sign"></i>
        Remove
      </template>
      <template ngSwitchDefault>
        <i class="glyphicon glyphicon-plus-sign"></i> Add
      </template>
    </button>
  </td>
 </tr>
</tbody>
</table>
```

Вновь созданная кнопка может запускать метод `toggleTask` в классе компонента, передавая ему в качестве аргумента объект `Task`, связанный с соответствующей итерацией директивы `NgFor`. С другой стороны, представленная реализация `NgSwitch` позволяет выводить на кнопках различные надписи и значки, в зависимости от состояния объекта `Task`, в любое заданное время.



Вновь созданные кнопки украшены шрифтовыми значками из семейства шрифтов Glyphicons. Значки входят в установленный ранее комплект Bootstrap CSS и не имеют никакого отношения к Angular 2. Вы можете не использовать их или поменять на другое семейство шрифтовых значков.

А сейчас выполните код и посмотрите на результат. Выглядит аккуратно, не так ли? Но с помощью Angular 2 список заданий можно сделать еще более функциональным.

Отображение изменений состояния в шаблонах

При выборе заданий в таблице было бы здорово получать визуальную подсказку, сколько Pomodoro-сеансов понадобится для их выполнения. Логика будет следующей:

- пользователь просматривает задания в таблице и, щелкая на них, выбирает те, которые собирается выполнить;
- каждый щелчок на строке приводит к изменению состояния низкоуровневого объекта `Task` и переключению его логического свойства `queued`;
- изменение состояния немедленно отображается выводом метки **queued** (в очереди) на элементе с соответствующим заданием;
- пользователю предоставляются информация о количестве требуемых Pomodoro-сеансов и оценка общей продолжительности их выполнения;
- над таблицей выводится ряд значков, отражающих суммарное количество Pomodoro-сеансов, необходимых для выполнения всех заданий.

Эта логика должна реагировать на изменение состояний множества объектов `Task`. Но благодаря собственной системе обнаружения изменений в Angular 2 отслеживание изменений состояний компонентов осуществляется очень просто.

Итак, наша первая задача – добавить в класс `TasksComponent` определение и отображение количества заданий, помещенных в очередь. Эта информация будет использоваться вне блока разметки компонента для отображения значков и суммарного количества времени для их выполнения.

Эту информацию будет предоставлять новое поле класса `queuedPomodoros`, а новый метод `updateQueuedPomodoros()` будет обновлять значение этого поля после создания экземпляра компонента или постановки задания в очередь. Кроме того, мы добавим отображение ключ/значение, которое позднее будет использовано для наглядного отображения в заголовке количества заданий в очереди с помощью фильтра `i18nPlural`:

```
class TasksComponent { today: Date;
  tasks: Task[];
  queuedPomodoros: number;
```

```
queueHeaderMapping: any = {
  '=0': 'No pomodoros',
  '=1': 'One pomodoro',
  'other': '# pomodoros'
};

constructor() {
  const TasksService: TasksService = new TasksService();
  this.tasks = taskService.taskStore;
  this.today = new Date();
  this.updateQueuedPomodoros();
}

toggleTask(task: Task): void { task.queued = !task.queued;
  this.updateQueuedPomodoros();
}

private updateQueuedPomodoros(): void {
  this.queuedPomodoros = this.tasks
    .filter((task: Task) => task.queued)
    .reduce((pomodoros: number, queuedTask: Task) => {
    return pomodoros + queuedTask.pomodorosRequired;
  }, 0);
  }
};
```

Метод `updateQueuedPomodoros()` использует встроенные методы массивов JavaScript – `Array.filter()` и `Array.reduce()` – для построения списка заданий в очереди из оригинальной коллекции свойства `tasks`. Метод `reduce`, примененный к результирующему массиву, вернет общее количество требуемых Pomodoro-сеансов. После определения количества Pomodoro-сеанов для заданий в очереди нужно соответствующим образом обновить шаблон. Откройте `pomodoro-tasks.html` и добавьте следующий HTML-фрагмент непосредственно перед элементом `<h4>Tasks backlog</h4>`:

```
<div>
<h3>
  {{ queuedPomodoros | i18nPlural:queueHeaderMapping }}
  for today
  <span class="small" *ngIf="queuedPomodoros > 0">
    (Estimated time: {{ queuedPomodoros * 25 }})
  </span>
</h3>
 </div>
 <h4>Tasks backlog</h4>
 <!-- остальная часть шаблона остается без изменений -->
```

Предыдущий блок отображает информативный заголовок, даже когда в очереди нет ни одного задания. Полученное значение включено в шаблон и используется для вычисления количества минут для выполнения всех Pomodoro-сеансов.



Продолжительность каждого Pomodoro-сеанса жестко зафиксирована в шаблоне. В идеале это значение следовало бы извлекать из переменной или из глобального параметра. Но не волнуйтесь, в следующих главах это будет исправлено.

Сохраните изменения, перезагрузите страницу и попробуйте переключить некоторые элементы заданий в таблице, чтобы убедиться, что сведения изменяются в режиме реального времени. Интересно, не правда ли?

Встраивание дочерних компонентов

А теперь начнем создание компонента крошечных значков с изображением помидора, который затем добавим внутрь компонента `TasksComponent`. Этот новый компонент будет отображать уменьшенную версию большого изображения помидора и использоваться для вывода в шаблоне количества Pomodoro-сеансов, необходимых для выполнения заданий в очереди, как было описано выше. Благодаря этому мы познакомимся с деревьями компонентов, которые более подробно будут рассмотрены в *главе 5 «Создание приложений с помощью компонентов Angular 2»*. А сейчас просто добавьте следующий класс компонента перед классом `TasksComponent`, созданным ранее.

Компонент будет экспортировать общедоступное свойство `task`, которому можно присвоить объект `Task`. Объект `Task` будет использоваться для создания копий изображения, количество которых равно количеству Pomodoro-сеансов для выполнения задания, что определено в свойстве `pomodorosRequired`. Копирование будет реализовано с помощью директивы `NgFor`.

Вставьте следующий блок кода в файл `pomodoro-tasks.ts` перед классом компонента TasksComponent:

```
@Component({
  selector: 'pomodoro-task-icons',
  template: `<img *ngFor="let icon of icons"
                  src="/assets/img/pomodoro.png" width="50">`
})
class TaskIconsComponent implements OnInit {
  @Input() task: Task;
```

```
  icons: Object[] = [];

  ngOnInit() {
    this.icons.length = this.task.pomodorosRequired;
    this.icons.fill({ name: this.task.name });
  }
}
```

Новый класс `TaskIconsComponent` имеет довольно простую реализацию с интуитивно понятным селектором, соответствующим имени класса в верблюжьей нотации, и шаблоном, копирующим тег `<img>` столько раз, сколько объектов находится в свойстве-массиве `icons` класса контроллера, которое заполняется с помощью метода `fill` объекта `Array` (метод `fill` присваивает всем элементам массива скалярное значение, переданное в качестве аргумента) внутри метода `ngOnInit()`. Подождите минутку, как это? Не следует ли реализовать цикл заполнения массива свойства `icons` в конструкторе?

Этот метод является одним из обработчиков событий жизненного цикла, которые более подробно рассматриваются в следующей главе. Причина, почему массив `icons` заполняется именно здесь, а не в конструкторе, заключается в необходимости инициализировать все свойства, перед тем как запустить цикл `for` . В противном случае значение `task` будет затребовано слишком рано, и в результате будет возвращено значение `undefined`.



Необходимость включения метода `ngOnInit()` в класс контроллера диктуется интерфейсом `OnInit`, который реализуется классом. Этот метод вызывается только после проверки привязок всех входных свойств. Обзор событий жизненного цикла приводится в *главе 5 «Создание приложений с помощью компонентов Angular 2»*.

Новый компонент должен иметь возможность обратиться к своему родителю. Для этого добавим ссылку на класс `component` в свойство `directives` декоратора `TasksComponent`:

```
@Component({
  selector: 'pomodoro-tasks',
  directives: [TaskIconsComponent],
  styleUrls: ['pomodoro-tasks.css'],
  templateUrl: 'pomodoro-tasks.html'
})
```

Помните, ранее мы говорили, что компоненты, по сути, являются директивами, снабженными пользовательскими представлениями? Если это так, значит, свойство `directives` любого компонента можно

использовать при вложении в него другого компонента. Этим и объясняется использование здесь свойства directives.

Наш следующий шаг – включение элемента <pomodoro-task-icons> в шаблон TasksComponent. Вернемся в pomodoro-tasks.html и изменим код внутри условного блока, отображаемого, когда значение queuedPomodoros больше нуля:

```
<div>
  <h3>
    {{ queuedPomodoros | i18nPlural:queueHeaderMapping }} for today
    <span class="small" *ngIf="queuedPomodoros > 0">
      (Estimated time: {{ queuedPomodoros * 25 }})
    </span>
  </h3>
  <p>
    <span *ngFor="let queuedTask of tasks">
      <pomodoro-task-icons [task]="queuedTask"
        (mouseover)="tooltip.innerText = queuedTask.name"
        (mouseout)="tooltip.innerText = 'Mouseover for details'">
      </pomodoro-task-icons>
    </span>
  </p>
  <p #tooltip *ngIf="queuedPomodoros > 0">Mouseover for details</p>
</div>
<h4>Tasks backlog</h4>
<!-- остальная часть шаблона остается без изменений -->
```

Но это еще не все, кое-что можно улучшить. К сожалению, размер значка жестко зафиксирован в шаблоне TaskIconsComponent, что затрудняет использование этого компонента в других контекстах, где может потребоваться другой размер. Мы могли бы исправить этот недостаток, добавив в класс TaskIconsComponent общедоступное входное свойство size, и привязать его к шаблону компонента, чтобы иметь возможность менять размер изображения:

```
@Component({
  selector: 'pomodoro-task-icons',
  template: `<img *ngFor="let icon of icons"
                  src="/assets/img/pomodoro.png"
                  width="{{size}}">`
})
class TaskIconsComponent implements OnInit {
  @Input() task: Task;
  icons: Object[] = [];
```

```
  @Input() size: number;

  ngOnInit() {
    ...
  }
}
```

Теперь осталось лишь изменить реализацию `pomodoro-tasks.html`, определив нужный размер:

```
<span *ngFor="let queuedTask of tasks">
  <pomodoro-task-icons
    [task]="queuedTask"
    size="50"
    (mouseover)="tooltip.innerText = queuedTask.name"
    (mouseout)="tooltip.innerText = 'Mouseover for details'">
    </pomodoro-task-icons>
</span>
```

Обратите внимание, что атрибут `size` не заключен в скобки, поскольку привязка жестко зафиксирована. Если бы потребовалось связать свойство с переменной компонента, атрибут следовало бы объявить как `[size]="{{mySizeVariable}}"`.

Давайте подытожим сделанное.

- Мы добавили новый элемент DOM, который выводится только при наличии Pomodoro-заданий в очереди.
- Реализовали отображение заголовка, сообщающего о количестве Pomodoro-заданий, ожидающих выполнения, связав свойство `queuedPomodoros` в элементе `H3` модели DOM, а также об оценке общего времени в минутах через выражение `{{ queuedPomodoros*25 }}`.
- Директива `NgFor` позволяет выполнить обход массива `tasks`, в каждой итерации которой выводится новый элемент `<pomodoro-task-icons>`.
- В каждой итерации цикла объект модели `Task`, представленный с помощью ссылки `queuedTask`, подключается к входному свойству `task` элемента `<pomodoro-task-icons>`.
- Кроме того, к элементу `<pomodoro-task-icons>` подключается дополнительный обработчик событий мыши, ссылаясь на абзац, отмеченный локальной ссылкой `#tooltip`. Таким образом, при наведении указателя мыши на значок Pomodoro текст под строкой значков будет отображать имя соответствующего Pomodoro-задания.

Дополнительно мы обеспечили возможность настройки размера значка, отображаемого элементом <pomodoro-task-icons>, через свойство компонента. Теперь значки Pomodoro обновляются в режиме реального времени при смене информации в таблице. Однако возникли новые проблемы. Во-первых, компоненты значков Pomodoro отображаются для всех заданий, а не только для помещенных в очередь. С другой стороны, общая оценка времени содержит общее количество минут, требующихся на выполнение всех Pomodoro-заданий, поставленных в очередь, и эта информация станет бессмысленной при добавлении большего количества Pomodoro-заданий в план работы.

Пришло время исправить это. Призовем на помощь пользовательские фильтры!

Создание собственных фильтров

Фильтры и их назначение в экосистеме Angular уже рассматривались выше, сейчас же будет описан процесс разработки набора собственных фильтров для осуществления пользовательских преобразований при привязке данных.

Анатомия пользовательских фильтров

Фильтры определяются просто. Во-первых, необходимо импортировать декоратор Pipe из основной библиотеки Angular и создать новый класс, отмеченный этим декоратором. Этот новый класс должен иметь имя, соответствующее параметрам декоратора, и реализовать интерфейс PipeTransform.

Реализация класса также довольно проста. Она включает обязательный метод transform, требуемый интерфейсом PipeTransform, который возвращает данные определенного типа (обычно их тип соответствует типу входных данных фильтра), и два параметра. В первом передаются сами входные данные, а второй, необязательный обобщенный аргумент (обобщенные аргументы в TypeScript были описаны *главе 2 «Введение в TypeScript»*), содержит параметры для настройки фильтра в представлении:

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'myPipeName',
  pure: false // необязательно, по умолчанию true
})
```

```
class MyPipe implements PipeTransform {
  transform(value: any, ...args: any[]): any {
    // Здесь выполняется преобразование входного значения
    return something;
  }
}
@Component({
  selector: 'my-selector',
  pipes: [MyPipe],
  template: '<p>{{ myVariable | myPipeName: "bar" }}</p>'
})
class myComponent {
  myVariable: string = 'Foo';
}
```



В предыдущем примере создан фильтр `MyPipe`, предназначенный для выполнения некоторого преобразования входных данных, в соответствии с параметрами, заданными при его применении, который будет определен ниже.

По аналогии с пользовательскими директивами, в свойстве `pipes` компонента должны быть явно перечислены реализуемые пользовательские фильтры.

Относительно необязательного свойства `pure` в декораторе `Pipe` следует пояснить, что фильтры не отслеживают изменения состояния. Это означает, что экземпляр фильтра будет повторно выполнен только при изменении его аргументов. Другими словами, при преобразовании входных данных фильтры используют не оригинальные входные данные, а обрабатывают их копии. Если позднее оригинальные данные изменятся, эти изменения не будут отражены в представлении. К счастью, имеется возможность включить отслеживание состояний фильтров с помощью логического свойства `pure`. Если присвоить ему значение `false`, фильтр будет следить за состоянием преобразуемых значений и выполнять заложенное в него выражение многократно, как только система обнаружения изменений Angular определит, что исходные данные изменились.

Применим эту концепцию на практике, создав пару пользовательских фильтров для нашего компонента.

Пользовательские фильтры для форматирования времени

Отслеживание общего количества минут с ростом числа заданий затрудняется, поскольку для большего удобства это количество следует

выводить в виде часов и минут. Фильтр pomodoroFormattedTime будет реализован с помощью класса FormattedTimePipe, чей метод transform получает общее количество минут и возвращает строку (что иллюстрирует отсутствие необходимости возврата фильтрами данных того же типа, что они получают на входе) в удобном для оценки длительности формате:

```
@Pipe({
  name: 'pomodoroFormattedTime'
})
class FormattedTimePipe implements PipeTransform {
  transform(totalMinutes: number): string {
    let minutes: number = totalMinutes % 60;
    let hours: number = Math.floor(totalMinutes / 60);
    return `${hours}h:${minutes}m`;
  }
}
```



Еще раз подчеркну, что в соответствии с соглашениями об именовании фильтров реализующие их классы компонентов должны иметь имя, совпадающее с именем фильтра, плюс суффикс Pipe. Здесь разница состоит в том, что селектор фильтра задан в верблюжьей нотации и с префиксом pomodoro. Почему же имя класса контроллера фильтра не соответствует его селектору? Обычно к именам пользовательских фильтров и директив добавляется пользовательский префикс, чтобы избежать конфликтов с селекторами фильтров и директив от сторонних производителей.

Не забывайте, что объявление пользовательских фильтров не делает их доступными в шаблонах, они должны быть явно объявлены в свойстве pipes декоратора каждого компонента, который их использует. В данном случае это компонент TasksComponent:

```
@Component({
  selector: 'pomodoro-tasks',
  directives: [PomodoroIconComponent],
  pipes: [FormattedTimePipe],
  styleUrls: ['pomodoro-tasks.css'],
  templateUrl: 'pomodoro-tasks.html'
})
class PomodoroTasksComponent {
  // Реализация класса остается неизменной
}
```

И наконец, следует внести изменения в разметку HTML шаблона pomodoro-tasks.html, добавив форматирование времени в соответствии с EDT:

```
<span class="small">
(Estimated time: {{queuedPomodoros*25 | pomodoroFormattedTime}})
</span>
```

Теперь перезагрузите страницу и переключите некоторые из заданий. Предполагаемое время их выполнения должно корректно отображаться в часах и минутах.

Фильтрация данных с помощью пользовательских фильтров

Отображаемые в данный момент значки Pomodoro соответствуют всем заданиям в коллекции, возвращаемой службой заданий, без фильтрации тех, что отмечены как находящиеся в очереди. Фильтры поддерживают удобную возможность сопоставления, преобразуя и сокращая привязываемые данные, что позволяет использовать их для фильтрации в цикле `NgFor` и отображения только заданий, отмеченных как включенные в очередь.

Логика довольно проста: поскольку свойство `tasks` является массивом объектов `Task`, можно воспользоваться методом `Array.filter()`, чтобы извлечь только те объекты `Task`, чьи свойства `queued` имеют значение `true`. Можно шагнуть еще дальше и добавить в фильтр логический аргумент, определяющий, что он должен фильтровать: задания из очереди или задания, не находящиеся в очереди. Следующая реализация вновь иллюстрирует применение соглашения об именовании селекторов и классов:

```
@Pipe({
  name: 'pomodoroQueuedOnly',
  pure: false
})
class QueuedOnlyPipe implements PipeTransform {
  transform(task: Task], ...args: any[]): Task[] {
    return tasks.filter((task: Task) => {
      return task.queued === args[0];
    });
  }
}
```

Реализация довольно проста, поэтому не будем вдаваться в ее подробности. Но обратите внимание на значение `false` в свойстве `pure` фильтра. Как вы помните, свойство `tasks` – это коллекция объектов, обладающих состояниями, длина и содержимое которой изменяются, когда пользователь переключает задания в таблице. По этой причине

следует обязать фильтр использовать систему обнаружения изменений в Angular, чтобы его вывод проверялся на каждом шаге цикла, независимо от изменения входных данных. Установка свойства pure декоратора фильтра в false позволяет добиться желаемого.

Теперь просто добавьте этот фильтр в свойство pipes компонента:

```
@Component({
  selector: 'pomodoro-tasks',
  directives: [PomodoroIconComponent],
  pipes: [FormattedTimePipe, QueuedOnlyPipe],
  styleUrls: ['pomodoro-tasks.css'],
  templateUrl: 'pomodoro-tasks.html'
})
class PomodoroTasksComponent {
  // Реализация класса остается неизменной
}
```

и отфильтруйте задания, не включенные в очередь, в блокt NgFor, в файле pomodoro-tasks.html:

```
<span *ngFor="#queuedTask of tasks | pomodoroQueuedOnly: true">
  <pomodoro-task-icons
    [task]="queuedTask"
    (mouseover)="tooltip.innerText = queuedTask.name"
    (mouseout)="tooltip.innerText = 'Mouseover for details'">
  </pomodoro-task-icons>
</span>
```

Обратите внимание, что фильтр настроен как pomodoroQueuedOnly: true. Замена значения логического параметра на false позволит перечислить Pomodoro-задания, не включенные в очередь.

Сохраните проделанную работу, перезагрузите страницу и переключите некоторые из заданий. Как видите, итоговый пользовательский интерфейс соответствующе реагирует на текущие изменения, и значки Pomodoro выводятся только для заданий, включенных в очередь.

Создание собственных директив

Пользовательские директивы дают огромные возможности. Количество вариантов их применения настолько велико, что для их описания потребуется отдельная книга.

Директивы позволяют придавать элементам DOM дополнительные функции. Если директива содержит вложенный шаблон, она становится компонентом. Другими словами, компоненты являются

директивами Angular с представлениями, но имеется возможность разрабатывать директивы без вложенных представлений, которые будут применяться к уже существующим элементам, позволяя связывать их HTML-содержимое и стандартное поведение с директивой. Это относится как к компонентам Angular, так и к директивам, имеющим доступ к шаблонам и пользовательским атрибутам, а при необходимости и к событиям.

Анатомия пользовательских директив

Директивы объявляются и реализуются просто. Необходимо лишь импортировать класс `Directive` для придания функциональности декоратора сопровождающему классу контроллера:

```
import { Directive } from '@angular/core';
```

Затем следует определить класс контроллера с декоратором `Directive`, где определяются селектор директивы, входные и выходные свойства (если требуются), дополнительные события, применяемые к вмещающему элементу, и встраиваемые имена поставщиков для директив, конструкторы которых требуют специфических типов при создании экземпляров с помощью инструмента внедрения зависимостей Angular 2 (подробнее об этом рассказывается в *главе 5 «Создание приложений с помощью компонентов Angular 2»*):

```
@Directive({
  selector: '[selector]',
  inputs: ['inputPropertyName'],
  outputs: ['outputPropertyName'],
  host: {
    '(event1)': 'onMethod1($event)',
    '(target:event2)': 'onMethod2($event)',
    '[prop]': 'expression',
    'attributeName': 'attributeValue'
  },
  providers: [MyCustomType]
})
class myDirective {
  @Input() otherInputPropertyName: any;
  @Output() otherOutputPropertyName: any;

  constructor(myCustomType: MyCustomType) {
    // реализация...
  }
}
```

Свойства и декораторы, такие как `selector`, `@Input()` или `@Output()` (а также `inputs` и `outputs`), будут кратко описаны при обзоре спецификации декоратора компонента. Ниже перечисляются варианты объявления селекторов, но это не полное описание всех возможностей:

- `element-name`: выбор по имени элемента;
- `.class`: выбор по имени класса;
- `[attribute]`: выбор по имени атрибута;
- `[attribute=value]`: выбор по имени и значению атрибута;
- `not(sub_selector)`: выбор, только если элемент не соответствует указанному `sub_selector`;
- `selector1, selector2`: выбор, если соответствует `selector1` или `selector2`.

В дополнение к этому для получения доступа из директивы используется параметр `host`, который определяет события, действия, свойства и атрибуты, относящиеся к вмещающему элементу (то есть элементу, с которым работает директива). Благодаря этому параметру можно привязать обработчики действий к контейнерному компоненту или любому другому целевому элементу, например `window`, `document` или `body`. В этом случае, определяя привязки директив, можно использовать в ссылках две удобные локальные переменные:

- `$event`: текущий объект события;
- `$target`: источник события. Источник события может быть элементом DOM или директивой Angular.

Помимо привязки к событиям, можно изменять определенные свойства вмещающего компонента. Для этого нужно связать с выражением любое конкретное свойство, заключенное в скобки, определив в параметре `host` директивы пару ключ/значение.

Необязательный параметр `host` может также определять статические атрибуты для применения к вмещающему элементу, если это еще не сделано. Это удобный способ внедрения HTML-свойств с помощью вычисляемых значений.

Команда Angular также предусмотрела пару удобных декораторов для более наглядного объявления привязок и обработчиков прямо в коде, например:

```
@HostBinding('[class.valid]')
isValid: boolean; // Вмещающий элемент получит атрибут
                  // class="valid", если 'isValid' равно true.
```

```
@HostListener('click', ['$event'])

onClick(e) {

  // Эта функция будет вызвана, когда в
  // компоненте произойдет событие 'click'.

}
```

В следующих главах мы подробнее рассмотрим интерфейс настройки директив и компонентов и особое внимание уделим управлению жизненным циклом и упрощению внедрения зависимостей в директивы. А сейчас давайте создадим простую, но мощную директиву, которая в корне изменит подход к отображению и управлению пользовательским интерфейсом.

Создание подсказки для заданий с помощью пользовательской директивы

Применим на практике некоторые из параметров пользовательских директив. Займемся выводом текстовых подсказок при наведении мыши на значки Pomodoro. Для этого подключим пару событий к элементу `<pomodoro-task-icons>`. Хотя это действующее решение, оно дает несколько громоздкий вывод и затрудняет повторное использование. В какой-то момент может потребоваться применить ту же привязку `[task]` в других местах, чтобы использовать те же удобные всплывающие подсказки. Реализуем это решение в виде директивы, чтобы упростить его применение к любым элементам с атрибутом `[task]`, а не только к элементу `<pomodoro-task-icons>`. Директива определяет входные свойства для ссылки на те же самые свойства и целевой элемент, который будет использоваться как заполнитель. Если атрибут отсутствует, директива не будет ничего делать, но и не вызовет исключений. При его наличии директива привяжет обработчики событий `mouseover` и `mouseout` к указанному элементу (`<pomodoro-task-icons>` в данном примере). Эти обработчики будут изменять текст внутри элемента DOM с помощью локальной ссылки на свойство заполнителя. Перед изменением исходное значение будет сохраняться для повторного использования, когда указатель мыши покинет элемент.

Все описанное выше реализуется следующей директивой, которая должна быть размещена в файле `pomodoro-tasks.ts` перед компонентами:

```
@Directive({ selector: '[task]' })
class TaskTooltipDirective {
```

```
  private defaultTooltipText: string;
  @Input() task: Task;
  @Input() taskTooltip: any;

  @HostListener('mouseover') onMouseOver() {
    if(!this.defaultTooltipText && this.taskTooltip) {
      this.defaultTooltipText = this.taskTooltip.innerText;
    }
    this.taskTooltip.innerText = this.task.name;
  }
  @HostListener('mouseout') onMouseOut() {
    if(this.taskTooltip) {
      this.taskTooltip.innerText = this.defaultTooltipText;
    }
  }
}
```

Обратите внимание на селектор `[task]`. Его использование позволило не менять логику элемента `<pomodoro-task-icons>` и не создавать нового селектора. Это возможно, но у этого примера другие цели. Нам нужно придать характерное поведение любому элементу DOM или компоненту с атрибутом `[task]`, используя присоединенные к нему данные. Именно потому данная директива будет воздействовать на все элементы, обладающие этим свойством, которое можно определить как входное свойство в реализации самой директивы. Затем необходимо предоставить способ настройки текста подсказки в элементе, определив входное свойство `taskTooltip`.

Как было описано в предыдущем разделе, с помощью декоратора `@HostListener()` можно подключить к директиве обработчики событий, возникающих во вмещающем компоненте. Здесь события `mouseover` и `mouseout` вызывают смену текста подсказки с предварительным сохранением текущего текста.

Чтобы увидеть директиву в действии, добавим ее поддержку в декоратор `PomodoroTasksComponent`:

```
@Component({
  selector: 'pomodoro-tasks',
  directives: [PomodoroIconComponent, TaskTooltipDirective],
  pipes: [FormattedTimePipe, pomodoroQueuedOnlyPipe],
  styleUrls: ['pomodoro-tasks.css'],
  templateUrl: 'pomodoro-tasks.html'
})
class PomodoroTasksComponent {
```

```
  // Дальнейшее без изменений
}
```

Затем внесем изменения в шаблон pomodoro-tasks.html:

```
<p>
  <span *ngFor="#queuedTask of tasks | pomodoroQueuedOnly: true">
    <pomodoro-task-icons
      [task]="queuedTask"
      [taskTooltip]="tooltip"
      size="50">
    </pomodoro-task-icons>
  </span>
</p>
<p #tooltip>Mouseover for details</p>
```

Одной из самых поразительных особенностей данного примера является весьма малый объем изменений, необходимых для придания элементам новой функциональности и возможности их повторного использования.

Внеся последние изменения, перезагрузите браузер, переключите любое задание, наведите курсор на значок вновь выведенного Pomodoro-задания и... вуаля!

Несколько слов об именах пользовательских директив и фильтров

Общее соглашение, ориентированное на многократное использование, рекомендует добавлять пользовательский префикс перед селекторами. Это позволит избежать конфликтов с селекторами из других библиотек, используемых в проекте. То же относится и к фильтрам, как было отмечено в разделе с описанием процесса создания первого пользовательского фильтра.

Здесь в качестве пользовательского префикса использован префикс pomodoro, и он будет применяться далее во всей книге, но я все же рекомендую использовать для селекторов ваших собственных директив и фильтров более короткий и узнаваемый префикс.

В конечном счете можно использовать любое соглашение, главное, чтобы оно предотвращало конфликты имен. Применение для этого пользовательского префикса, безусловно, является самым простым решением.

Итоги

На этом этапе можно с уверенностью утверждать, что вы знаете практически все необходимое для создания компонентов Angular 2, являющихся колесами и мотором всех приложений, использующих Angular 2. Будущие главы в основном посвящены архитектуре приложений, внедрению зависимостей в дереве компонентов, службам данных, применению нового маршрутизатора Angular для отображения и сокрытия компонентов по требованию, управлению пользовательским вводом и аутентификации.

В этой главе были рассмотрены базовые принципы разработки с помощью Angular 2 и достаточно подробно описан синтаксис шаблонов, программные интерфейсы компонентов, основанные на свойствах и событиях, инкапсуляция представлений, фильтры и директивы. Теперь можно переходить к новым задачам, от умения писать компоненты к их использованию для разработки больших приложений с применением соответствующих методик и рациональной архитектуры. Все это будет рассмотрено в следующей главе.

Глава 5

Создание приложений с помощью компонентов Angular 2

Мы достигли той точки в нашем путешествии, когда с уверенностью можем приступить к разработке сложного приложения с компонентами, вложенными в другие компоненты, образующими своеобразное дерево. Однако размещение логики всех компонентов в единственном файле – не самое удачное решение. Это очень скоро сильно затруднит поддержку приложения и сведет на нет преимущества, которые может привнести механизм управления зависимостями Angular.

В этой главе мы познакомимся с архитектурами приложений, основанных на деревьях компонентов, и посмотрим, как новый механизм внедрения зависимостей Angular 2 помогает объявлять и использовать зависимости с минимальными усилиями и оптимальным результатом.

Эта глава охватывает следующие темы:

- оптимальная организация структуры каталогов и соглашения об именовании;
- различные подходы к внедрению зависимостей;
- добавление зависимостей в пользовательские типы;
- переопределение глобальных зависимостей во всем дереве компонентов;

- взаимодействие с вмещающим компонентом;
- обзор жизненного цикла директив.

Введение в деревья компонентов

Современные веб-приложения, основанные на веб-компонентной архитектуре, часто образуют древовидную иерархию, в которой главный компонент верхнего уровня (обычно помещаемый внутрь главного HTML-файла) выступает в роли глобального контейнера для дочерних компонентов, которые, в свою очередь, содержат другие вложенные компоненты, и т. д.

Этот подход имеет очевидные преимущества. Во-первых, он не противоречит принципу многократного использования, позволяя повторно использовать компоненты в дереве компонентов, не прилагая к этому особых усилий. Во-вторых, достигнутая изолированность упрощает проектирование, разработку и обслуживание больших приложений. Он помогает сосредоточиться на одной из частей пользовательского интерфейса, затем обернуть его функциональность новыми уровнями абстракции, и так до тех пор, пока не будет обернуто все приложение, снизу доверху.

К разработке веб-приложения можно подойти с другого конца, начав с наиболее общей функциональности, двигаться вниз ко все более мелким фрагментам пользовательского интерфейса и функциональности, которые в итоге станут веб-компонентами. Последний подход более популярен при создании архитектуры на основе компонентов. В оставшейся части книги мы будем применять именно его, приняв за основу обобщенную архитектуру, изображенную на рис. 5.1.

Рис. 5.1

Для большей наглядности в этой главе будет применяться код из предыдущих глав, который мы поместим в иерархию компонентов. В результирующем приложении будет предусмотрено место для всех вспомогательных классов и моделей, необходимых для формирова-

ния инструмента Pomodoro. Это станет хорошей возможностью познакомиться с особенностями механизма внедрения зависимостей, встроенного в Angular 2.

Общие соглашения для масштабируемых приложений

Справедливости ради следует отметить, что уже решен ряд общих задач, стоящих перед веб-разработчикам при создании как малых, так и больших современных приложений. Поэтому имеет смысл определить архитектуру, которая выделит упомянутые выше задачи в отдельные папки, содержащие медиа-ресурсы и модули с общим кодом.

На момент написания книги согласованные шаблоны организации каталогов проекта основываются на идее структурирования файлов по функциям или контексту. Иногда может потребоваться совместное использование некоторых сущностей двумя контекстами, и в этом нет ничего плохого (до тех пор, пока это не стало слишком частым явлением в проекте, что означает серьезные просчеты в проектировании). На рис. 5.2 приведена такая схема применительно к предыдущей работе над компонентами проекта Pomodoro.

```
.
├── возможности заданий
│   ├── модель задания
│   ├── службы заданий
│   ├── табличный компонент заданий
│   └──
Компонент Pomodoro-задания
│   └── директива подсказок для заданий
├── функция таймера
│   └── компонент таймера
├── администрирование
│   ├── служба аутентификации
│   ├── компонент регистрации
│   └── компонент редактора
└── общее
    ├── совместно используемые компоненты
    ├── совместно используемые фильтры
    ├── совместно используемые директивы
    ├── глобальные модели и службы
    └── общие медиа-ресурсы
```

Рис. 5.2

Как видите, на первом шаге определяются различные *возможности*, необходимые приложению, причем подразумевается, что каждая из них должна иметь собственный смысл в отрыве от других. После этого для каждой возможности создается отдельная папка. Затем каждая из папок заполняется компонентами, директивами, фильтрами, моделями и службами, которые обеспечивают данную возможность. Не следует забывать о принципах инкапсуляции и повторного использования при определении набора возможностей.

Если количество файлов, необходимых для обеспечения любой из возможностей, превышает некоторый логический максимум, следует разделить эти файлы на разные папки по *типу*. Допустим, выяснилось, что *возможности заданий* чрезмерно разрослись и соответствующая папка содержит уже более 30 файлов с компонентами, директивами, фильтрами, службами, тестами и т. д. Это затрудняет идентификацию модулей кода, поэтому следует добавить дополнительный слой или разбить файлы по типам, как показано на рис. 5.3.

```
├── возможности заданий
│   ├── компоненты/
│   │   └── файлы компонентов...
│   ├── директивы/
│   │   ├── директива X
│   │   └── директива Y
│   ├── фильтры/
│   │   └── файлы фильтров...
│   ├── модели/
│   │   └── модели...
│   └── службы/
│       └── службы...
├── функция таймера
│   └── компонент таймера
├── администрирование
│   ├── служба аутентификации
│   ├── компонент регистрации
│   └── компонент редактора
└── общее
    ├── ...
    └── прочее
```

Рис. 5.3

Как видите, очень удобно, когда на одном уровне в проекте имеются папки с файлами реализации возможностей и присутствуют вложенные папки для разделения файлов по типам. Максимальное количество выбирается индивидуально, но здравый смысл подсказывает, что при наличии более 12–15 модулей в папке имеет смысл перейти к дополнительному уровню вложенности, основанному на типах. Также можно объединить несколько возможностей путем добавления дополнительных уровней вложенности, основанных на наборе возможностей верхнего контекста, разделив их реализацию на дочерние деревья, разбиваемые по типам, как показано на рис. 5.3.

```
.
├── возможности заданий
│   ├── компоненты и шаблоны заданий
│   ├── возможность редактирования заданий/
│   │   └── компоненты и шаблоны для редактирования заданий
│   ├── возможности списка заданий/
│   │   ├── компоненты/
│   │   │   └── компоненты и шаблоны списка заданий
│   │   └── фильтры/
│   │       └── фильтры для списка заданий
│   └── возможность создания отчетов о заданиях/
│   └── службы...
...
```

Рис. 5.4

Соглашение об именовании файлов и модулей

Каждая из папок возможностей вмещает широкий диапазон файлов, поэтому требуется согласованность имен для предотвращения конфликтов и облегчения поиска различных модулей.

В следующем списке обобщаются существующие соглашения, предлагаемые сообществом.

- Каждый файл должен содержать только один модуль кода. Каждый компонент, директива, служба, фильтр и т. д. должен размещаться в отдельном файле. Это улучшает организацию кода.
- Имена файлов и каталогов задаются в `нижнем-регистре-с-дефисом-между-словами`.
- К именам файлов с компонентами, директивами, фильтрами и службами должен добавляться суффикс, определяющий их тип, например файлу `video-player.ts` следует дать имя `video-player.component.ts`.

- Имена файлов внешних HTML-шаблонов или внешних таблиц стилей CSS компонентов должны совпадать с именами файлов самих компонентов, включая суффикс. Файл компонента `video-player.component.ts` могут дополнять файлы `video-player.component.css` и `video-player.component.html`.
- Селекторы директив и имена фильтров задаются в `верблюжьейНотации`, в то время как селекторы компонентов – `нижнем-регистре-с-дефисом-между-словами`. Кроме того, настоятельно рекомендуется добавлять собственный префикс, чтобы избежать конфликтов имен с другими библиотеками компонентов. Например, имя компонента video-player следует преобразовать в `<vp-video-player>`, где `vp-` (от *video-player*) является пользовательским префиксом.
- Модулям присваиваются описательные имена в `СтилеЯзыкаPascal` (с первой прописной буквой) с добавлением типа. Например, модуль `VideoPlayerComponent` должен содержать только компонент. Пользовательский префикс, используемый в селекторах (в данном примере `vp-`), не должен быть частью имени модуля.

Модели и интерфейсы требуют особого внимания. В зависимости от архитектуры приложения типы моделей будут иметь большую или меньшую актуальность. Такие архитектуры, как MVC, MVVM, Flux или Redux, по-разному рассматривают модели и придают им различное значение. В конечном счете шаблон проектирования архитектуры определяет подход к моделям и их именованию. В эту книгу не будут включены конкретные рекомендации, хотя она и содержит описание применения интерфейсных моделей и создания для них модулей.

Обеспечение непрерывной масштабируемости с помощью фасадов или баррелей

Каждый компонент и общий контекст бизнес-логики приложения предназначен для интеграции с другими фрагментами собираемой картинки. Клиентов каждой дочерней предметной области не должна беспокоить внутренняя структура самой дочерней предметной области. Например, если возможность *timer*, используемая двумя дюжинами компонентов и директив, перенесена в папку другого уровня, внешние потребители этой функции должны сохранить возможность пользоваться ею.

Это может быть достигнуто с помощью фасадных модулей, скрывающих внутреннюю структуру кода и экспортирующих неизменный набор точек входа для внешних потребителей.

В предыдущем примере расположение файлов *возможности заданий* в одной папке изменяется при разделении файлов на папки по типам. Что делать, если некоторые из ее компонентов использовались в другом месте в приложении? Не беспокойтесь! Здесь поможет определение следующего фасадного модуля:

APP/TASKS/TASKS.TS

```
import TaskComponent from './task.component';
import TaskDetailsComponent from './task-details.component';

export {
  TaskComponent,
  TaskDetailsComponent
}
```

После этого вы сможете импортировать любой из этих компонентов в любом месте приложения, например:

APP/EXAMPLE/EXAMPLE.TS

```
import { TaskDetailsComponent } from '../tasks/tasks'
```

Что, если *возможности заданий* постоянно разрастаются и нужно разделить их на несколько папок? Достаточно обновить пути в фасадном модуле `app/tasks/tasks.ts`, и код клиента `app/example/example.ts` останется неизменным.

На жаргоне Angular такой шаблон проектирования называется *баррель* (barrel). Цитата из документации Angular:

> ***Баррель*** *– это библиотечный модуль Angular, логически объединяющий модули с одинаковым назначением, например модули* `Component` *и* `Directive`.

Имейте это в виду, чтобы не запутаться, наткнувшись на данный термин. Кроме того, баррели обычно группируются и распространяются в составе более крупных пакетов, называемых *связками* (bundles). Примером может служить связка `angular2/bundles/router.js` для создания маршрутизаторов. В этой книге мы не будем создавать связки, но при реализации маршрутизации HTTP-соединений и анимации широко используются связки, поставляемые в составе Angular 2 .

Внедрение зависимостей в Angular 2

По мере роста и развития приложений однажды наступает момент, когда программным сущностям потребуется создавать экземпляры

других объектов, которые в программировании называются *зависимостями*. Процесс передачи такой зависимости нуждающемуся в ней клиенту называется *внедрением*, и он предполагает участие в процессе еще одной программной сущности, называемой *инструментом внедрения зависимостей*. Инструмент внедрения зависимостей берет на себя ответственность за создание экземпляров и начальную загрузку требуемых зависимостей, подготавливая их к использованию сразу после успешного внедрения. Это очень важно, поскольку клиент ничего не знает о порядке создания экземпляра зависимости, а осведомлен только о реализуемом зависимостью интерфейсе.

Angular 2 снабжен первоклассным механизмом внедрения зависимостей, упрощающим задачу их экспортирования в любые сущности приложения, будь то компоненты, директивы, фильтры или любые другие пользовательские службы и сторонние объекты. В самом деле, как будет показано далее в этой главе, любая сущность может воспользоваться поддержкой внедрения зависимостей, реализованной в Angular 2. Перед тем как углубиться в эту тему, рассмотрим, как механизм внедрения зависимостей решает проблему адресации.

Предположим, что имеется компонент для воспроизведения музыки, работа которого основывается на объекте списка воспроизведения, позволяющем транслировать музыку пользователям:

```
import { Component } from '@angular/core';
import { Playlist } from './playlist';

@Component({
  selector: 'music-player',
  templateUrl: './music-player.component.html'
})
class MusicPlayerComponent {
  playlist: Playlist;

  constructor() {
    this.playlist = new Playlist();
  }
}
```

Тип `Playlist` может быть универсальным классом, возвращающим случайный список песен или что-то еще. Это сейчас не важно, поскольку здесь имеет значение только то, что он необходим сущности `MusicPlayerComponent` для работы. К сожалению, приведенная выше реализация означает, что эти сущности тесно связаны между собой,

так как компонент создает список воспроизведения *внутри* своего конструктора. Это сводит на нет возможность изменения, переопределения или имитации класса `Playlist` при необходимости. Также это предполагает создание нового объекта `Playlist` для каждого экземпляра `MusicPlayerComponent`. В определенных ситуациях это нежелательно, особенно если в приложении требуется использовать единственный экземпляр `Playlist` и таким способом сохранять состояние списка воспроизведения.

Система внедрения зависимостей помогает решить эти проблемы, предлагая несколько шаблонов, и среди них – шаблон **внедрения через конструктор**. Чтобы реализовать его, предыдущий фрагмент кода можно изменить так:

```
@Component({
  selector: 'music-player',
  templateUrl: './music-player.component.html'
})
class MusicPlayerComponent {
  playlist: Playlist;

  constructor(playlist: Playlist) {
    this.playlist = playlist;
  }
}
```

Теперь экземпляр `Playlist` будет создан за пределами компонента. С другой стороны, компонент `MusicPlayerComponent` требует приготовить нужный объект до создания самого компонента, чтобы его можно было внедрить через конструктор. Этот подход предоставляет возможность переопределения или имитации объекта при возникновении такой необходимости.

Примерно так работает внедрение зависимостей и, в частности, внедрение через конструктор. Однако какое отношение это имеет к Angular 2? Неужели работа механизма внедрения зависимостей в Angular заключается лишь в ручном создании экземпляров классов и внедрении их через конструктор? Очевидно, что нет, поскольку экземпляры компонентов практически никогда не создаются вручную (кроме случаев, связанных с написанием модульных тестов). В Angular имеется собственный фреймворк внедрения зависимостей, который, кстати, может использоваться в качестве автономного фреймворка в других приложениях.

Фреймворк предлагает настоящий инструмент внедрения зависимостей, способный анализировать аннотации параметров в кон-

структоре и возвращать одиночный экземпляр типа для каждой из зависимостей, что дает возможность использовать его непосредственно в реализации класса, как показано в предыдущем примере. Сам инструмент внедрения зависимостей не занимается созданием экземпляров для каждой из зависимостей, полагаясь на список поставщиков (providers), зарегистрированных в процессе начальной загрузки приложения. Каждый из этих поставщиков фактически *обеспечивает* сопоставление типов, помеченных как зависимости приложения. Всякий раз, когда сущность (скажем, компонент, директива или служба) определяет параметр в своем конструкторе, инструмент внедрения зависимостей находит соответствующий ему тип в наборе зарегистрированных для данного компонента поставщиков. Если совпадение не будет найдено, поиск делегируется родительскому компоненту поставщика и продолжается до тех пор, пока не будет найден соответствующий тип или достигнут компонент верхнего уровня. Если поиск поставщика завершится неудачей, Angular 2 возбудит исключение.



Последнее утверждение не всегда верно, поскольку имеется возможность отметить зависимость в конструкторе с помощью декоратора параметра `@Optional`, сообщающего фреймворку Angular 2, что вместо исключения должно быть внедрено значение `null`, если поставщик не найден. Верхний компонент не является последней инстанцией при поиске поставщика, поскольку имеется еще возможность объявлять глобальные зависимости в функции `bootstrap()`, как это показано ниже.

Всякий раз, когда поставщик находит тип, соответствующий параметру, он возвращает одиночный экземпляр этого типа, который затем будет внедрен инструментом внедрения зависимостей. Справедливости ради отмечу, что поставщик представляет собой не просто набор пар ключ/значение для заранее зарегистрированных типов, а фабрику, которая создает экземпляры этих типов, а также создает зависимости для каждой из этих зависимостей, осуществляя рекурсивную обработку зависимостей при создании экземпляра.

Таким образом, вместо создания экземпляра объекта `Playlist laylist` можно сделать следующее:

```
import { Component } from '@angular/core';
import { Playlist } from './playlist';

@Component({
  selector: 'music-player',
```

```
  templateUrl: './music-player.component.html',
  providers: [Playlist]
})
class MusicPlayerComponent {
  constructor(public playlist: Playlist) {}
}
```

Свойство `providers` декоратора `@Component` является тем местом, где можно зарегистрировать зависимости на уровне компонента. С этого момента перечисленные типы станут доступны для внедрения в конструкторе данного компонента и в его дочерних компонентах, как это будет продемонстрировано позднее.

Внедрение зависимостей в дереве компонентов

Как уже упоминалось, поиск поставщика ведется снизу вверх, пока не будет найдено совпадение. Приведем более наглядный пример. Пусть имеется компонент музыкального приложения, который содержит в своем свойстве `directives` (и, следовательно, в своем шаблоне) компонент с коллекцией всех загруженных мелодий, который также содержит в своем свойстве `directives` и в шаблоне компонент, предназначенный для воспроизведения любой мелодии из библиотеки (рис. 5.5).

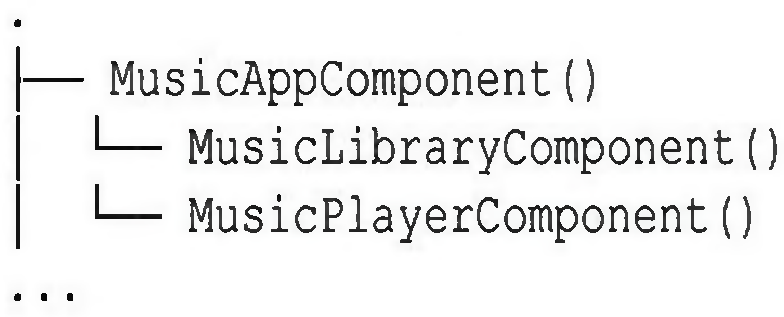


Рис. 5.5

Компоненту воспроизведения требуется упомянутый выше объект `Playlist`, поэтому он объявлен как параметр, аннотированый типом `Playlist`.

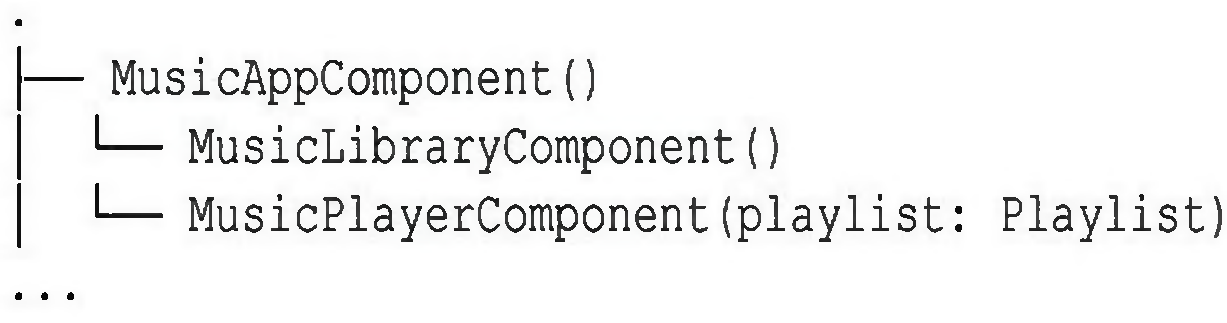


Рис. 5.6

При создании экземпляра объекта `MusicPlayerComponent` механизм внедрения зависимостей фреймворка Angular обходит параметры

конструктора компонента, уделяя особое внимание их аннотациям типов. Далее он проверяет свойство `provider` декоратора компонента. Для этого код компонента должен выглядеть следующим образом:

```
@Component({
  selector: 'music-player',
  providers: [Playlist]
})
class MusicPlayerComponent {
  constructor(public playlist: Playlist) {}
}
```

Но что, если необходимо повторно использовать тип `Playlist` в других компонентах в том же дереве? Допустим, что тип `Playlist` реализует функциональные возможности, которые требуются различным компонентам во всем приложении. Следует ли регистрировать тип в свойстве `providers` каждого из них? К счастью, нет, поскольку Angular 2 предполагает такое требование и обеспечивает перекрестное внедрение зависимостей в дереве компонентов.



В предыдущем разделе было упомянуто, что поиск поставщика ведется вверх от компонента. Это связано с тем, что каждый компонент обладает собственным, специфичным для него инструментом внедрения зависимостей. Однако этот инструмент внедрения зависимостей является дочерним экземпляром инструмента внедрения зависимостей родительского компонента (и т. д.), то есть приложение, использующее Angular 2, имеет не один инструмент внедрения зависимостей, а, так сказать, множество экземпляров одного и того же инструмента.

Нам необходимо распространить внедрение объекта `Playlist` на другие компоненты в дереве компонентов, сделав его простым и позволяющим многократное использование. Зная, что компоненты выполняют поиск поставщика, начиная с самого себя, а затем передают запрос инструменту внедрения зависимостей родительского компонента, можно переадресовать задачу регистрации поставщика родительскому компоненту или даже самому верхнему родительскому компоненту, сделав, таким образом, зависимость доступной для внедрения всем дочерним компонентам более низкого уровня. Тогда имеет смысл зарегистрировать объект `Playlist` прямо в `MusicAppComponent`, независимо от необходимости его применения в реализации этого компонента:

```
@Component({
  selector: 'music-app',
  providers: [Playlist],
  directives: [MusicLibraryComponent],
  template: '<music-library></music-library>'
})
class MusicAppComponent {}
```

Благодаря этому отпадает необходимость регистрировать зависимость в дочерних компонентах. Поскольку она уже зарегистрирована в родительском компоненте `MusicAppComponent` и нет нужды регистрировать ее еще раз.

```
@Component({
  selector: 'music-library',
  directives: [MusicPlayerComponent],
  template: '<music-player></music-player>'
})
class MusicLibraryComponent {}
```

И наконец, мы достигли компонента воспроизведения музыки, в котором тоже не требуется регистрировать тип `Playlist` в свойстве `providers`. Фактически этот компонент вообще не имеет свойства `providers`. Оно больше не требуется, поскольку нужный тип уже зарегистрирован где-то выше в иерархии компонентов, что сразу же делает его доступным для всех дочерних компонентов, независимо от того, где они находятся.

```
@Component({
  selector: 'music-player'
})
class MusicPlayerComponent {
  constructor(public playlist: Playlist) {}
}
```

Итак, зависимости доступны для внедрения вниз по иерархии компонентов, и поиск поставщиков выполняется компонентами путем просмотра их собственных зарегистрированных поставщиков и передачи запроса вверх по дереву компонентов. Но что, если потребуется ограничить область внедрения или поиска?

Ограничение внедрения зависимостей вниз по дереву компонентов

В предыдущем примере музыкального приложения регистрация типа `Playlist` в коллекции `providers` компонента делает нужную за-

висимость доступной для всех дочерних компонентов. Но иногда требуется ограничить внедрение зависимостей только директивой или компонентом, которые непосредственно следуют за определенным компонентом в иерархии. Это можно сделать, зарегистрировав тип в свойстве `viewProviders` декоратора компонента, а не в свойстве `providers`, как делалось ранее. В предыдущем примере можно ограничить внедрение зависимости `Playlist` только одним уровнем:

```
@Component({
  selector: 'music-app',
  viewProviders: [Playlist],
  directives: [MusicLibraryComponent],
  template: '<music-library></music-library>'
})
class MusicAppComponent {}
```

Здесь фреймворку Angular 2 указывается, что поставщик `Playlist` должен быть доступен лишь инструментам внедрения зависимостей директив и компонентов, расположенных в представлении `MusicAppComponent`, но не в дочерних компонентах. Такой способ применяется для ограничения области внедрения только их представлениями.

Ограничение поиска поставщика

Подобно тому, как можно ограничить область внедрения зависимостей, можно ограничить и поиск зависимостей только непосредственно родительским уровнем. Для этого к таким параметрам-зависимостям нужно применить декоратор `@Host()`, поиск поставщика которых следует ограничить:

```
import { Component, Host } from '@angular/core';

@Component({
  selector: 'music-player'
})
class MusicPlayerComponent {
  constructor(@Host() playlist: Playlist) {}
}
```

В предыдущем примере инструмент внедрения зависимостей компонента `MusicPlayerComponent` будет искать тип `Playlist` в коллекции `providers` родительского компонента (`MusicLibraryComponent` в данном случае) и остановится, вызвав исключение, потому что `Playlist` не будет возвращен инструментом внедрения зависимостей родитель-

ского элемента (если не декорировать его параметром декоратора `@Optional()`).

Переопределение поставщиков в иерархии инструментов внедрения зависимостей

До настоящего момента механизм ведрения зависимостей фреймворка Angular использовал аннотации типов для поиска требуемого типа и возвращал его из любого набора поставщиков, доступных в иерархии компонентов. Однако в некоторых случаях может потребоваться переопределить экземпляр класса, соответствующий этому типу, чтобы получить более специализированный тип. Фреймворк Angular предоставляет специальные инструменты для переопределения поставщиков или даже для реализации фабрик, которые будут возвращать экземпляр класса, не обязательно совпадающего с оригинальным типом.

Мы не будем вдаваться в детали всех случаев использования, а рассмотрим лишь простой пример. Допустим, что объект `Playlist` должен быть доступен в дереве компонентов для использования в различных сущностях приложения. А что делать, если директива `MusicAppComponent` содержит другой компонент, чьим дочерним директивам требуется более специализированная версия объекта `Playlist`? Давайте переосмыслим пример:

```
.
├── MusicAppComponent()
│   ├── MusicChartsComponent()
│   │   └── MusicPlayerComponent()
│   └── MusicLibraryComponent()
│   └── MusicPlayerComponent()
...
```

Рис. 5.7

Это несколько надуманный пример, но он помогает понять идею переопределения зависимостей. Объект экземпляра `Playlist` доступен непосредственно от верхнего компонента и далее вниз. Директива `MusicChartsComponent` является специализированным компонентом, работающим только с хитами, и, следовательно, ее проигрыватель должен воспроизводить только хиты, несмотря на то что он использует тот же компонент `MusicLibraryComponent`. Необходимо обеспечить получение каждым проигрывателем нужного

объекта списка воспроизведения, и это может быть сделано на уровне компонента MusicChartsComponent с помощью переопределения экземпляра объекта, соответствующего типу Playlist. Следующий пример демонстрирует этот способ, основанный на использовании функции provide:

```
import { Component, provide } from '@angular/core';
import { Playlist } from './playlist';
import { TopHitsPlaylist } from './top-hits-playlist';

@Component({
  selector: 'music-charts',
  directives: [MusicPlayerComponent],
  template: '<music-player></music-player>',
  providers: [provide(Playlist, { useClass: TopHitsPlaylist })]
})
class MusicChartsComponent {}
```

Функция provide() создает поставщика, соответствующего типу, переданному ей в виде первого аргумента (в данном примере это Playlist), и реализацию, управляемую вторым аргументом, который в данном случае задает использование типа TopHitsPlaylist.

Можно дополнительно изменить блок кода и задействовать viewProviders, чтобы (если это требуется) дочерние сущности по-прежнему получали экземпляр класса Playlist, а не TopHitsPlaylist. Кроме того, можно приложить дополнительные усилия и использовать фабрику для создания экземпляра конкретного объекта, в зависимости от некоторых требований. Следующий пример возвращает экземпляры различных объектов для типа Playlist, в зависимости от проверки значения логической переменной condition:

```
@Component({
  selector: 'music-charts',
  directives: [MusicPlayerComponent],
  template: '<music-player></music-player>',
  providers: [
    provide(Playlist, {
      useFactory: () => {
        if(condition) {
          return new TopHitsPlaylist();
        } else {
          return new Playlist();
        }
      }
```

```
    })
  ]
})
class MusicChartsComponent {}
```

Можно ли, основываясь на псевдокоде из предыдущего примера, обеспечить более приемлемый поток логики при использовании функции useFactory? Получается, что сигнатура метода может принимать аргументы, которые работают подобно зависимостям в конструкторе любой сущности Angular. Просто следует указать фреймворку Angular тип каждого аргумента в лямбда-функции useFactory, объявив их в свойстве deps:

```
@Component({
  selector: 'music-charts',
  directives: [MusicPlayerComponent],
  template: '<music-player></music-player>',
  providers: [
    ConditionalService,
    provide(Playlist, {
      useFactory: (conditionalService) => {
        if(conditionalService.isTopHits) {
          return new TopHitsPlaylist();
        } else {
          return new Playlist();
        }
      },
      deps: [ConditionalService]
    })
  ]
})
class MusicChartsComponent {}
```

В предыдущем примере был внедрен экземпляр объекта мнимого класса ConditionalService, экспортирующий логическое свойство isTopHits, которое сообщает об используемом списке воспроизведения. Имейте в виду, что типы должны быть зарегистрированы в свойстве providers текущего компонента или любого из его родителей.

Расширенная поддержка инструментов внедрения зависимостей для пользовательских сущностей

Директивы и компоненты требуют, чтобы зависимости были проанализированы, найдены и внедрены. Но и другим сущностям, например

служебным классам, часто требуется подобная функциональность. В данном примере классу `Playlist` может быть необходим HTTP-клиент, связывающей его с третьей стороной для загрузки песни. Внедрение такой зависимости должно быть столь же простым, как объявление аннотированных зависимостей в конструкторе класса, и должен существовать инструмент внедрения зависимостей для получения экземпляра объекта через поставщика класса или любого другого доступного поставщика.

При осмыслении предыдущего утверждения выясняется противоречие: пользовательские классы и службы не принадлежат дереву компонентов. Следовательно, у них отсутствует встроенный или родительский инструмент внедрения зависимостей. В них нельзя даже объявить свойство `providers`, поскольку классы этих типов не имеют декораторов `@Component` и `@Directive`. Рассмотрим пример:

```
class Playlist {
  songs: string[];
  constructor(songsService: SongsService) {
    this.songs = songsService.fetch();
  }
}
```

Можно попробовать ввести параметр `songsService` в конструктор класса `Playlist`, понадеявшись на его анализ механизмом внедрения зависимостей фреймворка Angular 2 при создании экземпляра этого класса в процессе внедрения его в `MusicPlayerComponent`. К сожалению, единственное, что будет получено в результате, – это следующее исключение:

```
Cannot resolve all parameters for Playlist(?). Make sure they all have valid
type or annotations.

(Невозможно разрешить все параметры для Playlist. Убедитесь, что все типы
правильно аннотированы.)
```

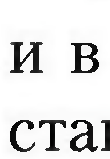
Это сообщение не соответствует действительности, поскольку все параметры конструктора класса `Playlist` должным образом аннотированы, не так ли? Как уже упоминалось, механизм внедрения зависимостей во фреймворке Angular разрешает зависимости, анализируя типы параметров конструктора. При этом он использует подготовленные заранее метаданные. Класс каждой Angular-сущности декорируется метаданными из декоратора как побочный результат TypeScript-компиляции деталей конфигурации декоратора. Однако

зависимости, которые требуют других зависимостей, не имеют ни декоратора, ни метаданных. Это легко исправить с помощью декоратора `@Injectable()`, который сделает эти служебные классы видимыми для механизма внедрения зависимостей:

```
import { Injectable } from '@angular/core';

@Injectable()
class Playlist {
  songs: string[];
  constructor(songsService: SongsService) {
    this.songs = songsService.fetch();
  }
}
```

Вам придется привыкнуть к использованию этого декоратора в определениях служебных классов, поскольку они часто нуждаются в зависимостях, не связанных с деревом компонентов.



Возьмите в привычку декорирование всех служебных классов декоратором `@Injectable()`, независимо от наличия зависимостей в их конструкторах. Это позволит избежать ошибок и исключений в дальнейшем, когда служебный класс вырастет и ему потребуются зависимости.

Инициализация приложения с помощью функции bootstrap()

Как уже упоминалось в этой главе, поиск зависимостей ведется от компонента вверх. Но это не совсем верно, поскольку на работу механизма внедрения зависимостей влияет дополнительный фактор – функция `bootstrap()`.

Функция `bootstrap()` используется для запуска приложения, и в первом аргументе ей передается корневой компонент, который становится основанием для дерева компонентов приложения. Однако функция `bootstrap()` поддерживает и второй аргумент – массив поставщиков, определяющий зависимости, доступные инструментам внедрения зависимостей во всем дереве. Но это не лучший способ задания зависимостей, поскольку он обеспечивает доступность поставщиков в самом приложении, препятствуя инкапсуляции и повторному использованию компонентов. Кроме того, функция `bootstrap()` специфична для каждой из платформ.

Где же тогда объявлять зависимости приложения? Всегда используйте корневой компонент верхнего уровня. Аргумент с постав-

щиками в вызове функции `bootstrap` следует использовать только при необходимости переопределить существующих поставщиков на уровне приложения, там, где используется, например, функция `provide()`.

Имейте в виду, что с помощью функции `bootstrap()` можно задать несколько корневых компонентов, включив в приложение несколько вызовов `bootstrap()` с разными корневыми компонентами. Каждый из этих компонентов будет иметь собственный набор инструментов внедрения зависимостей, не связанных между собой.

Переключение между режимами разработки и эксплуатации

Приложения на основе Angular 2 по умолчанию загружаются и инициализируются в *режиме разработки*. В этом режиме среда выполнения Angular 2 выводит предупреждающие сообщения и сообщения об исключениях в консоль браузера. Это очень полезно при отладке приложения, но эти сообщения не должны отображаться при работе приложения в режиме эксплуатации. Могу вас обрадовать: режим разработки можно сменить на менее многословный режим эксплуатации. Обычно это действие выполняется перед начальной загрузкой приложения:

```
import { bootstrap } from '@angular/platform-browser-dynamic';
import { enableProdMode } from 'angular/core';
import AppComponent from './app.component';

enableProdMode();

bootstrap(AppComponent, []);
```

Включение встроенного профилировщика обнаружения изменений

Подключив инструменты отладки Angular, можно получить доступ к ним из консоли браузера. Для этого следует импортировать функцию `enableDebugTools`, которая специфична для платформы браузера, и вызвать ее, как только будет получен экземпляр компонента, который нуждается в профилировании:

```
import { bootstrap } from '@angular/platform-browser-dynamic';
import { enableDebugTools } from '@angular/platform-browser';
import { ComponentRef } from 'angular/core';
import AppComponent from './app.component';
```

```
// Функция bootstrap() возвращает объект promise со
// ссылкой на компонент начальной загрузки
bootstrap(AppComponent, []).then((ref: ComponentRef) => {
  enableDebugTools(ref);
});
```

После вызова функции `enableDebugTools()` требуется выполнить следующие действия, чтобы получить доступ к профилировщику.

1. Откройте инструменты разработчика в браузере и переключитесь в режим консоли.
2. Введите `ng.profiler.timeChangeDetection({record: true})` и нажмите клавишу **Enter**.
3. Среда выполнения Angular 2 приступит к поиску изменений в цикле и выведет среднюю продолжительность одной итерации для текущего состояния пользовательского интерфейса. Одновременно с обнаружением изменений будет вестись регистрация данных профилирования центрального процессора.

Будем надеяться, что в будущем инструменты отладки станут более функциональными. Оставайтесь на связи и просматривайте официальную документацию.

Подготовка структуры каталогов приложения Pomodoro

В предыдущих главах и разделах этой главы были рассмотрены различные подходы и методики компоновки приложений, использующих Angular 2. От соглашений об именовании до структуры файлов и папок. Сейчас мы применим полученные знания на практике, чтобы привести интерфейсы, компоненты, директивы, фильтры и службы в соответствие с архитектурой Angular 2, рекомендуемой наиболее популярными соглашениями сообщества.

К концу этой главы будет получена окончательная компоновка приложения, размещающая все сделанное до настоящего момента в приведенную на рис. 5.8 архитектуру сайта.

Схема на рис. 5.8 помогает понять логику проекта. Теперь мы распределим все сущности приложения по двум основным контекстам: *таймер* и *список заданий*. Каждая возможность может включать широкий диапазон различных компонентов, фильтров, директив и служб. Внутренняя реализация каждой возможности непрозрачна для других возможностей или контекстов. Контекст каждой возмож-

ности имеет фасад, который экспортирует часть ее функциональности (то есть один или несколько компонентов), каждый контекст подпитывает контекст самого верхнего уровня или всего приложения. Все другие аспекты (внутренние службы или директивы) скрыты от остальной части приложения.

```
.
├── app/
│   ├── shared/
│   │   ├── assets/ ← Глобальные CSS-файлы и файлы изображений
│   │   ├── directives/
│   │   ├── interfaces/
│   │   ├── pipes/
│   │   ├── services/
│   │   └── shared.ts ← фасад для "общего" контекста
│   ├── tasks/
│   │   ├── (связанные с заданиями компоненты и директивы)
│   │   └── tasks.ts ← фасад контекста "заданий"
│   ├── timer/
│   │   ├── (связанные с таймером компоненты и директивы)
│   │   └── timer.ts ← фасад для контекста "таймер"
│   │
│   ├── app.component.ts ← корневой компонент приложения
│   └── main.ts ← здесь загружается корневой компонент
│
├── index.html
├── package.json
├── tsconfig.json
└── typings.json
```

Рис. 5.8

Справедливости ради следует отметить, что трудно четко нарисовать линию, ограничивающую то, что принадлежит одному или другому конкретному контексту. Некоторые аспекты функциональности, например директивы или фильтры, могут использоваться во всем приложении. Поэтому не имеет смысла изолировать их в пределах конкретного контекста. Для таких случаев существует *общий контекст*, где хранится весь код, предназначенный для многократного использования на уровне приложения, кроме медиа-файлов, таких как таблицы стилей или изображения, не зависящих от компонентов.

Главный файл `app.component.ts` содержит и экспортирует корневой компонент приложения, который объявляет и регистрирует соб-

ственный инструмент внедрения зависимостей, необходимый его дочерним компонентам. Как уже упоминалось, все приложения, использующие Angular 2, должны иметь, по крайней мере, один корневой компонент, инициализируемый функцией `bootstrap()`. Фактически эта операция выполняется в файле `main.ts`, который запускается в файле `index.html`.

Определение компонента или группы связанных компонентов внутри контекста обеспечивает возможность повторного использования и инкапсуляции. Единственным компонентом, который тесно связан с приложением, является корневой компонент, функциональность которого обычно достаточно ограничена и заключается в отображении дочерних компонентов в своем шаблоне представления или выполнении роли маршрутизатора компонентов, как это будет продемонстрировано в последующих главах.

Последним фрагментом мозаики являются JSON-файлы с настройками компилятора TypeScript и `npm` и определениями типов. Поскольку механизм управления версиями фреймворка Angular 2 постоянно развивается, здесь не будет рассматриваться фактическое содержимое этих файлов. Нужно лишь знать их назначение, а особенности, такие как версии зависимостей, изменяются довольно часто, поэтому за информацией об их последних версиях лучше обратиться в репозиторий на GitHub. Однако файл `package.json` требует особых пояснений. Существует несколько общих соглашений и популярных заготовок проектов, ряд которых можно найти на официальном сайте Angular. Ниже приводится несколько команд `npm`, упрощающих процесс установки и подготовки среды разработки.

1. Перейдите на страницу https://github.com/deeleman/angular2-essentials и загрузите файлы `package.json`, `typings.json` и `tsconfig.json` в выбранную папку. Рекомендуем загрузить и файл `index.html`.
2. Откройте консоль в папке, где были сохранены файлы, и выполните команду `npm install`. Фреймворк Angular 2, все его зависимости и объявления типов будут загружены и установлены в рабочем каталоге проекта.
3. Затем можно выполнить в консоли команду `npm start`, активирующую компилятор TypeScript в режиме отслеживания, и локальный сервер, настроенный на обслуживание папки проекта. Рекомендуется создать файлы приложения перед выполнением этой команды, иначе будет возбуждено исключение.

Приведение приложения в соответствие с рекомендациями Angular 2

В этом разделе мы разобьем код, приведенный в главах 1, 3 и 4, на модули, по принципу **персональной ответственности**. Но особых изменений в самом коде это не вызовет, мы лишь выделим модули в отдельные файлы. Именно поэтому основное внимание будет уделено разделению, а не пояснениям к каждому из модулей, поскольку они уже были описаны ранее. В любом случае, если это потребуется, будет сделано отступление для пояснения вносимых изменений.

Начнем с создания структуры каталогов, которая была приведена в предыдущем разделе, и параллельно будем наполнять папки файлами.

Общий контекст

Общий контекст будет хранить модули, предназначенные для использования не в одном, а сразу в нескольких контекстах, при этом никак не зависящие от этих контекстов. Хорошим примером является изображение помидора, предназначенное для визуального оформления компонентов, которое должно храниться в папке `app/shared/assets/img` (кстати, поместите его туда).

Еще одним хорошим примером являются интерфейсы моделей данных, главным образом из-за того, что их схемы могут повторно использоваться в другом контексте. Например, когда в *главе 3 «Реализация свойств и событий компонентов»* был определен класс `QueuedOnlyPipe`, мы задействовали только свойство `queued` элементов в наборе записей. Имеет смысл рассмотреть вопрос о реализации интерфейса `Queued`, который в дальнейшем можно использовать для проверки типа в модулях, где присутствует данное свойство. Это позволит упростить многократное использование фильтров и сделать их независимыми от модели. Например:

APP/SHARED/INTERFACES/QUEUABLE.TS

```
interface Queueable {
  queued: boolean;
}
export default Queueable;
```



Обратите внимание на порядок действий, описанный выше: сначала определяется модуль, соответствующий этому блоку кода, а затем производится экспорт с пометкой `default`, чтобы можно было

импортировать его по имени в других местах. Интерфейсы следует экспортировать именно таким способом, но в остальной части книги модуль будет объявляться и экспортироваться в одном и том же операторе.

С помощью этого интерфейса можно безопасно модифицировать класс `QueuedOnlyPipe`, сделав его полностью независимым от интерфейса `Task`, что позволит многократно использовать его в любом контексте, где может потребоваться фильтровать набор записей, состоящий из элементов, реализующих интерфейс `Queued`, независимо от того, что они собой представляют. Например:

APP/SHARED/PIPES/QUEUED-ONLY.PIPE.TS

```
import { Pipe, PipeTransform } from 'angular/core';
import { Queueable } from '../shared';

@Pipe({
  name: 'pomodoroQueuedOnly', pure: false
})
export default class QueuedOnlyPipe implements PipeTransform {
  transform(
  queueableItems: Queueable[],
  ...args): Queueable[] {
    return queueableItems.filter((queueableItem: Queueable) => {
      return queueableItem.queued === args[0]
    });
  }
}
```

Как видите, каждый блок кода содержит один модуль. Этот блок соответствует соглашениям об именовании файлов в Angular 2, требующим указывать имя модуля в верблюжьей нотации, а также суффикс типа (в данном случае `.pipe`). В реализации ничего не меняется, за исключением аннотирования всех элементов, которые могут быть включены в очередь типом `Queuable` вместо `Task`, как это было ранее. Теперь фильтр может использоваться везде, где присутствует модель, реализующая интерфейс `Queued`.

Однако имеется еще кое-что, на что следует обратить внимание: импорт интерфейса `Queuable` выполнен не из его исходного местоположения, а из файла `shared.ts`, размещенного на верхнем уровне. Этот фасадный файл из контекста *shared* будет экспортировать все общедоступные модули не только внешним, но и внутренним клиентам, составляющим сам общий контекст. Для этого имеются веские

основания: если любой модуль в рамках контекста *shared* изменит свое местоположение, достаточно обновить фасад, чтобы любой другой элемент, ссылающийся на модуль в том же контексте, оставался неизменным, поскольку он использует модуль через фасад. Наступил удобный момент для наполнения первого фасада:

APP/SHARED/SHARED.TS

```
import Queueable from './interfaces/queueable';

export {
  Queueable
};
```

Как видите, фасады не содержат бизнес-логики и в самом простом случае лишь объединяют экспортируемые элементы в единственный оператор экспорта. Теперь, получив работоспособный интерфейс `Queuable` и фасад, можно создать другой интерфейс, который потребуется во всех последующих примерах, соответствующий сущности `Task`, вместе с фильтрами, и все это экспортировать через фасад:

APP/SHARED/INTERFACES/TASK.TS

```
import { Queueable } from '../shared';

interface Task extends Queueable {
  name: string;
  deadline: Date;
  pomodorosRequired: number;
}

export default Task;
```

Для реализации наследования интерфейса от другого интерфейса в TypeScript используется ключевое слово `extends` (а не `implements`). А теперь реализуем `FormattedTimePipe`:

APP/SHARED/PIPES/FORMATTED-TIME.PIPE.TS

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'pomodoroFormattedTime'
})
export default class FormattedTimePipe implements PipeTransform {
  transform(totalMinutes: number): string {
    let minutes: number = totalMinutes % 60;
```

```
    let hours: number = Math.floor(totalMinutes / 60);
    return `${hours}h:${minutes}m`;
  }
}
```

Очевидно, что оба модуля будут экспортироваться через фасад, как это делалось ранее.

Службы в общем контексте

Не существует однозначного правила, регламентирующего размещение служб. Некоторые полагают, что службы, являясь лишь поставщиками данных и логики, не должны зависеть от контекста применения – компонентов, директив или других служб. Они считают службы полноправными гражданами, реализация которых должна оформляться в виде отдельных проектов, чтобы упростить их повторное использование в других проектах. Другие предпочитают привязывать службы к контекстам, которым они принадлежат, если они применяются только в одном контексте, отдавая предпочтение инкапсуляции. В конечном счете все зависит от соотношения важности повторного использования и инкапсуляции и того, какие сущности фактически используют данные и логику этих служб.

В предыдущей главе была разработана служба данных для наполнения таблицы со списком заданий. Как мы увидим ниже, эта служба будет использоваться также в других контекстах приложения. Поэтому вынесем ее в общий контекст, экспортировав ее через фасад, как обычно:

APP/SHARED/SERVICES/TASK.SERVICE.TS

```
import { Injectable } from '@angular/core';
import { Task } from '../shared';

@Injectable()
export default class TaskService {
  public taskStore: Task[] = [];

  constructor() {
    const tasks = [
      {
        name: "Code an HTML Table",
        deadline: "Jun 23 2015",
        pomodorosRequired: 1
      }, {
        name: "Sketch a wireframe for the new homepage",
```

```
      deadline: "Jun 24 2016",
      pomodorosRequired: 2
    }, {
      name: "Style table with Bootstrap styles",
      deadline: "Jun 25 2016",
      pomodorosRequired: 1
    }, {
      name: "Reinforce SEO with custom sitemap.xml",
      deadline: "Jun 26 2016",
      pomodorosRequired: 3
    }
  ];

  this.taskStore = tasks.map(task => {
    return {
      name: task.name,
      deadline: new Date(task.deadline),
      queued: false,
      pomodorosRequired: task.pomodorosRequired
    };
  });
 }
}
```

Обратите внимание на импорт декоратора `Injectable()` и его применение к службе. Служба не содержит в своем конструкторе зависимостей, поэтому в других модулях, зависящих от этой службы, не возникнет проблем при ее объявлении в конструкторах. Тем не менее возьмите в привычку применение декоратора `@Injectable()` во всех службах, чтобы обеспечить поддержку внедрения зависимостей, на тот случай, когда они станут зависимыми от других поставщиков.

Настройка параметров приложения из центральной службы

В предыдущих главах в компонентах было жестко зафиксировано много значений, таких как надписи, длительность Pomodoro-сеанса, определение множественного числа в именах существительных и т. д. Иногда контексты следует конкретизировать и поместить в них всю информацию. В других случаях могут потребоваться большая гибкость и удобный способ изменения этих параметров на уровне приложения. В этом примере все i18n-фильтры и параметры можно будет определять с помощью центральной службы из общего контекста, экспортируемой, как обычно, через фасад `shared.ts`.

APP/SHARED/SERVICES/SETTINGS.SERVICE.TS

```
import { Injectable } from '@angular/core';

@Injectable()
export default class SettingsService {
  timerMinutes: number;
  labelsMap: any;
  pluralsMap: any;

  constructor() {
    this.timerMinutes = 25;
    this.labelsMap = {
      'timer': {
        'start': 'Start Timer',
        'pause': 'Pause Timer',
        'resume': 'Resume Countdown',
        'other': 'Unknown'
      }
    };
    this.pluralsMap = {
      'tasks': {
        '=0': 'No pomodoros',
        '=1': 'One pomodoro',
        'other': '# pomodoros'
      }
    }
  }
}
```

Обратите внимание, как экспортируются свойства, не зависящие от контекста, что фактически аналогично организации пространств имен по контекстам.

Еще лучше было бы разделить эту службу на две, по одной для каждого контекста, поместив их в соответствующие папки. По крайней мере, это касается отображений для интернационализации. Имейте в виду, что такие данные, как длительность Pomodoro-сеанса, будут использоваться в разных контекстах, как показано ниже в этой главе.

Создание фасадного модуля, включающего пользовательскую баррель поставщиков

С учетом всех последних изменений фасад `shared.ts` должен выглядеть следующим образом:

APP/SHARED/SHARED.TS

```
import Queueable from './interfaces/queueable';
import Task from './interfaces/task';

import FormattedTimePipe from './pipes/formatted-time.pipe';
import QueuedOnlyPipe from './pipes/queued-only.pipe';

import SettingsService from './services/settings.service';
import TaskService from './services/task.service';

export {
  Queueable,
  Task,

  FormattedTimePipe,
  QueuedOnlyPipe,

  SettingsService,
  TaskService
};
```

Фасад экспортирует типы, фильтры и поставщиков служб. При глобальном внедрении зависимостей в корневой компонент для большего удобства часто производится группировка служб (и директив, включаемых в ту же группу, что и компоненты) с присваиванием псевдонимов группам, обычно по имени контекста с добавлением суффикса `_PROVIDERS` и с использованием только прописных букв. Что касается фасада, прямо перед оператором экспорта можно добавить следующий фрагмент:

```
// операторы import выше без изменений
const SHARED_PIPES: any[] = [
  FormattedTimePipe,
  QueuedOnlyPipe
];

const SHARED_PROVIDERS: any[] = [
  SettingsService,
  TaskService
];

export {
  Queueable,
  Task,

  FormattedTimePipe,
  QueuedOnlyPipe,
  SHARED_PIPES,
```

```
  SettingsService,
  TaskService,
  SHARED_PROVIDERS
};
```

Таким способом, указав единственный псевдоним, можно зарегистрировать всех нужных поставщиков. То же относится к директивам и компонентам, где, как правило, для экспорта в фасаде контекста используется псевдоним вида {ИМЯ_КОНТЕКСТА.}_DIRECTIVES. Это дает возможность внедрить поддержку всех необходимых компонентов и директив контекста в другой компонент, указав единственный псевдоним. Если в дальнейшем такой контекст будет экспортировать больше компонентов и директив или изменятся имена логических модулей, это не приведет к необходимости добавлять или изменять имена модулей, регистрируемых в свойствах directives компонентов во всем приложении. Все это будет показано ниже в этой главе.

Создание компонентов

После оформления общего контекста можно заняться двумя другими контекстами: таймера и заданий. Имена контекстов достаточно точно определяют их функциональность. Папка каждого контекста будет содержать файлы компонентов, HTML-шаблонов, таблиц CSS и директив, необходимых для обеспечения функциональности, и фасадный файл, экспортирующий общедоступные компоненты.

Контекст таймера

Первый контекст – контекст **timer** – кажется наиболее простым. Он включает уникальный компонент с таймером, разработанным в предыдущих главах:

APP/TIMER/TIMER-WIDGET.COMPONENT.TS

```
import { Component, Input, OnInit } from '@angular/core';
import { SettingsService } from '../shared/shared';

@Component({
  selector: 'pomodoro-timer-widget',
  template: `
  <div class="text-center">
    <img src="/app/shared/assets/img/pomodoro.png">
    <h1> {{ minutes }}:{{ seconds | number: '2.0' }} </h1>
    <p>
      <button (click)="togglePause()" class="btn btn-danger">
      {{ buttonLabelKey | i18nSelect: buttonLabelsMap }}
```

```
      </button>
    </p>
  </div>`
})
export default class TimerWidgetComponent {
  minutes: number;
  seconds: number;
  isPaused: boolean;
  buttonLabelKey: string;
  buttonLabelsMap: any;

  constructor(private settingsService: SettingsService) {
    this.buttonLabelsMap = settingsService.labelsMap.timer;
  }

  ngOnInit(): void {
    this.resetPomodoro();
    setInterval(() => this.tick(), 1000);
  }

  resetPomodoro(): void {
    this.isPaused = true;
    this.minutes = this.settingsService.timerMinutes - 1;
    this.seconds = 59;
    this.buttonLabelKey = 'start';
  }

  private tick(): void {
    if (!this.isPaused) {
      this.buttonLabelKey = 'pause';

      if (--this.seconds < 0) {
        this.seconds = 59;
        if (--this.minutes < 0) {
          this.resetPomodoro();
        }
      }
    }

    togglePause(): void {
      this.isPaused = !this.isPaused;
      if (this.minutes < this.settingsService.timerMinutes ||
          this.seconds < 59) {
        this.buttonLabelKey = this.isPaused ? 'resume' : 'pause';
      }
    }
}
```

Как видите, реализация практически не отличается от той, что мы видели в *главе 1 «Создание самого первого компонента в Angular 2»*, за исключением инициализации компонента в начале жизненного цикла с помощью обработчика `OnInit`. Для улучшения обработки разных меток, соответствующих каждому из состояний таймера, здесь использован фильтр `i18nSelect`, получающий информацию о метках от поставщика `SettingsService`, внедряемого через аннотированный аргумент конструктора. Порядок регистрации поставщика будет описан далее в этой главе. Длительность Pomodoro-сеанса в минутах также извлекается из службы путем ее привязки к полю класса.

Компонент экспортируется через фасад, что избавляет клиентские компоненты от необходимости знать фактический путь и имя файла компонента:

APP/TIMER/TIMER.TS

```
import TimerWidgetComponent from './timer-widget.component';

const TIMER_DIRECTIVES: any[] = [
  TimerWidgetComponent
];

export {
  TIMER_DIRECTIVES,
  TimerWidgetComponent
};
```

Обратите внимание на псевдоним `TIMER_DIRECTIVES`, введенный для удобства на случай увеличения количества компонентов в будущем, что дает возможность пользоваться единой точкой доступа ко всем общедоступным компонентам и директивам.

Контекст заданий

Контекст **tasks** включает немного больше логики, так как в нем присутствуют два компонента и директива. Начнем с создания основного модуля для директивы `TaskTooltipDirective`:

APP/TASKS/TASK-TOOLTIP.DIRECTIVE.TS

```
import { Task } from '../shared/shared';
import { Input, Directive, HostListener } from '@angular/core';

@Directive({
  selector: '[task]'
})
```

```
export default class TaskTooltipDirective {
  private defaultTooltipText: string;
  @Input() task: Task;
  @Input() taskTooltip: any;

  @HostListener('mouseover')
  onMouseOver() {
    if(!this.defaultTooltipText && this.taskTooltip) {
      this.defaultTooltipText = this.taskTooltip.innerText;
    }
    this.taskTooltip.innerText = this.task.name;
  }
  @HostListener('mouseout')
  onMouseOut() {
    if(this.taskTooltip) {
      this.taskTooltip.innerText = this.defaultTooltipText;
    }
  }
}
```

Директива сохранила все оригинальные функции и просто импортирует необходимые встроенные типы Angular 2 и тип `Task`. А теперь посмотрим на `TaskIconsComponent`:

APP/TASKS/TASK-ICONS.COMPONENT.TS

```
import { Component, Input, OnInit } from '@angular/core';
import { Task } from '../shared/shared';

@Component({
  selector: 'pomodoro-task-icons',
  template: `<img *ngFor="let icon of icons"
    src="/app/shared/assets/img/pomodoro.png"
    width="{{size}}">`
})
export default class TaskIconsComponent implements OnInit {
  @Input() task: Task;
  @Input() size: number;
  icons: Object[] = [];

  ngOnInit() {
    this.icons.length = this.task.pomodorosRequired;
    this.icons.fill({ name: this.task.name });
  }
}
```

Пока все в порядке. А теперь перейдем к `TasksComponent`. Этот компонент требует приложить больше усилий, поскольку он использует

внешние HTML-шаблоны и таблицы стилей, которые почти полностью совпадают с созданными в *главе 4 «Расширение компонентов с помощью директив и конвейеров»*:

APP/TASKS/TASKS.COMPONENT.CSS

```
h3, p {
    text-align: center;
}
.table {
    margin: auto;
    max-width: 860px;
}
```

APP/TASKS/TASKS.COMPONENT.HTML

```
<div class="container text-center">
  <h3>
    {{ queuedPomodoros | i18nPlural: queueHeaderMapping }} for today
    <span class="small" *ngIf="queuedPomodoros > 0"> (Estimated time:
      {{ queuedPomodoros * timerMinutes | pomodoroFormattedTime }})
    </span>
  </h3>
<p>
  <span
    *ngFor="let queuedTask of tasks | pomodoroQueuedOnly: true">
    <pomodoro-task-icons [task]="queuedTask" [taskTooltip]="tooltip"
      size="50">
    </pomodoro-task-icons>
  </span>
</p>
<p #tooltip [hidden]="queuedPomodoros === 0"> Mouseover for details
</p>

<h4>Tasks backlog</h4>
<table class="table">
  <thead>
    <tr>
      <th>Task ID</th>
      <th>Task name</th>
      <th>Deliver by</th>
      <th>Pomodoros</th>
      <th>Actions</th>
    </tr>
</thead>
<tbody>
```

```
<tr *ngFor="let task of tasks; let i = index">
  <th scope="row">{{i}}
    <span *ngIf="task.queued" class="label label-info"> Queued
    </span>
  </th>
  <td>{{task.name | slice: 0:35 }}
    <span [hidden]="task.name.length < 35">...</span>
  </td>
<td>{{task.deadline | date: 'fullDate' }}
<span *ngIf="task.deadline < today" class="label label-danger">
  Due
</span>
</td>
<td class="text-center">{{task.pomodorosRequired}}</td>
<td>
  <button type="button" class="btn btn-default btn-xs"
    [ngSwitch]="task.queued" (click)="toggleTask(task)">
    <template [ngSwitchWhen]="false">
      <i class="glyphicon glyphicon-plus-sign"></i> Add
    </template>
    <template [ngSwitchWhen]="true">
      <i class="glyphicon glyphicon-minus-sign"></i> Remove
    </template>
    <template ngSwitchDefault>
      <i class="glyphicon glyphicon-plus-sign"></i> Add
    </template>
  </button>
</td>
</tr>
</tbody>
</table>
</div>
```

Найдите минутку для проверки правильности применения соглашений об именовании к внешним файлам компонентов, чьи имена должны соответствовать имени компонента, чтобы проще было определить принадлежность файлов в папке контекста с одноуровневой структурой. Кроме того, обратите внимание, что из шаблона было удалено главное изображение помидора, а длительность Pomodoro-сеанса в выражении привязки, вычисляющем длительность выполнения всех заданий в очереди, теперь не задается жестко зафиксированным числом, а извлекается из переменной `timerMinutes`. Присваивание значения этой переменной происходит в следующем классе компонента:

APP/TASKS/TASKS.COMPONENT.TS

```
import { Component, OnInit } from '@angular/core';
import TaskIconsComponent from './task-icons.component';
import TaskTooltipDirective from './task-tooltip.directive';
import {
  TaskService,
  SettingsService,
  Task,
  SHARED_PIPES
} from '../shared/shared';

@Component({
  selector: 'pomodoro-tasks',
  directives: [TaskIconsComponent, TaskTooltipDirective],
  pipes: [SHARED_PIPES],
  styleUrls: ['app/tasks/tasks.component.css'],
  templateUrl: 'app/tasks/tasks.component.html'
})
export default class TasksComponent implements OnInit {
  today: Date;
  tasks: Task[];
  queuedPomodoros: number;
  queueHeaderMapping: any;
  timerMinutes: number;

  constructor(
    private taskService: TaskService,
    private settingsService: SettingsService) {

    this.tasks = this.taskService.taskStore;
    this.today = new Date();
    this.queueHeaderMapping = settingsService.pluralsMap.tasks;
    this.timerMinutes = settingsService.timerMinutes;
  }

  ngOnInit(): void {
    this.updateQueuedPomodoros();
  }

  toggleTask(task: Task): void {
    task.queued = !task.queued;
    this.updateQueuedPomodoros();
  }

  private updateQueuedPomodoros(): void {
    this.queuedPomodoros = this.tasks
    .filter((task: Task) => task.queued)
```

```
    .reduce((pomodoros: number, queuedTask: Task) => {
      return pomodoros + queuedTask.pomodorosRequired;
    }, 0);
  }
};
```

Стоит отметить следующие аспекты реализации компонента `TasksComponent`:

- нам пришлось импортировать необходимый дочерний компонент и директиву. Если попытаться привязать их через фасад, мы получим неопределенные значения из-за циклической ссылки;
- нам не потребовалось импортировать все необходимые фильтры. В данном случае достаточно просто зарегистрировать псевдоним `SHARED_PIPES` в свойстве `pipes` декоратора компонента;
- для внедрения поставщиков `TaskService` и `SettingsService` в компонент был использован механизм внедрения зависимостей фреймворка Angular. Параметры конструктора с модификаторами доступа, куда выполняется внедрение, автоматически превращаются в закрытые свойства класса;
- в конце конструктор заполняет список заданий и устанавливает длительность Pomodoro-сеанса с помощью привязанных служб.

Наш последний шаг – экспортирование фасада данного контекста. Сразу отмечу, что не все в данном контексте предназначено для экспорта. Достаточно ограничиться экспортом только основного компонента контекста, опустив подчиненный компонент и директиву, поскольку весьма маловероятно, что внешнему компоненту потребуется использовать `TaskIconsComponent` в своем представлении. Но мы все же экспортируем директиву `TaskTooltipDirective`, так как ее реализация может использоваться в некоторых других компонентах, имеющих дело с входными свойствами `[task]`.

APP/TASKS/TASKS.TS

```
import TasksComponent from './tasks.component';
import TaskTooltipDirective from './task-tooltip.directive';

const TASKS_DIRECTIVES: any[] = [
  TasksComponent,
  TaskTooltipDirective
];

export {
```

```
  TASKS_DIRECTIVES,
  TasksComponent,
  TaskTooltipDirective
];
```

Проверьте соответствие соглашению об именовании псевдонимов при группировке компонентов и директив.

Определение корневого компонента

Теперь, когда готовы все функциональные контексты, пришло время определить корневой компонент, который будет служить стартовой точкой всего приложения и кластером компонентов с древовидной иерархией. Корневой компонент, как обычно, имеет минимальную реализацию. Основное его назначение заключается в регистрации поставщиков зависимостей, используемых на различных уровнях иерархии компонентов и для создания в его шаблоне представления экземпляров основных дочерних компонентов, которые в конечном итоге превратятся в ветви дочерних компонентов.

APP/APP.COMPONENT.TS

```
import { Component } from '@angular/core';
import { TIMER_DIRECTIVES } from './timer/timer';
import { TASKS_DIRECTIVES } from './tasks/tasks';
import { SHARED_PROVIDERS } from './shared/shared';

@Component({
  selector: 'pomodoro-app',
  directives: [TIMER_DIRECTIVES, TASKS_DIRECTIVES],
  providers: [SHARED_PROVIDERS],
  template: `
    <nav class="navbar navbar-default navbar-static-top">
      <div class="container">
        <div class="navbar-header">
          <strong class="navbar-brand">My Pomodoro App</strong>
        </div>
      </div>
    </nav>

    <pomodoro-timer-widget></pomodoro-timer-widget>
    <pomodoro-tasks></pomodoro-tasks>
  `
})
export default class AppComponent {}
```

Оцените, насколько удобнее импортировать и регистрировать псевдонимы в свойствах `providers` и `directives` декоратора компонента. Псевдоним `SHARED_PROVIDERS` заслуживает особого упоминания. Все поставщики, сгруппированные под его именем, доступны, начиная с корневого компонента и далее вниз по дереву, поэтому нет необходимости в его повторной регистрации, и состояние каждого поставщика будет одинаковым во всем приложении.

Единственное исключение можно было бы сделать для службы `TaskService` и регистрировать ее в компоненте определенного уровня. Это позволило бы создать новый экземпляр службы для этого компонента и всех его дочерних компонентов.

Начальная загрузка приложения

Теперь у нас есть полноценное приложение с разными контекстами, обернутыми корневым компонентом. Осталось только реализовать начальную загрузку приложения, импортировав корневой компонент и передав его функции `bootstrap()`:

APP/MAIN.TS

```
import { bootstrap } from '@angular/platform-browser-dynamic';
import AppComponent from './app.component';

bootstrap(AppComponent);
```

Этот файл необходимо импортировать в файле `index.html`, чтобы запустить весь процесс с помощью стандартного загрузчика модулей, такого как SystemJS или WebPack. В репозитории книги используется SystemJS, поэтому при рассмотрениии кода в этой главе ориентируйтесь на его применение. В файл `index.html` нужно поместить пользовательский элемент, соответствующий селектору корневого компонента, как показано ниже, где приводится содержимое файла `index.html`:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>My Angular 2 Pomodoro Application</title>

    <script src="node_modules/es6-shim/es6-shim.min.js"></script>

    <script src="node_modules/zone.js/dist/zone.js"></script>
    <script src="node_modules/reflect-metadata/Reflect.js"></script>
    <script src="node_modules/systemjs/dist/system.js"></script>
    <script src="node_modules/rxjs/bundles/Rx.js"></script>
```

```
<script src="systemjs.config.js"></script>
<script>
  System.import ('built/app/main')
  .then(null, console.error.bind(console));
</script>
<link rel="stylesheet"
href="node_modules/bootstrap/dist/css/bootstrap.min.css">

<base href="/">
  </head>

  <body>
    <pomodoro-app>Loading...</pomodoro-app>
  </body>

</html>
```

Собрав вместе все необходимые файлы, можно скомпилировать проект и посмотреть на полученный результат в окне браузера. Если вы загрузили файл `package.json` (и связанные с ним JSON-файлы) из репозитория книги, выполните команду `npm start` в окне терминала и можете наслаждаться результатом. Вы самостоятельно создали впечатляющее приложение Pomodoro!

Итоги

В этой главе был заложен фундамент для всех великолепных приложений, которые вы сможете создавать, опираясь на Angular 2. Реализация управления зависимостями в Angular 2 является одной из изюминок этого фреймворка, которая значительно экономит время. Архитектура приложения, базирующаяся на деревьях компонентов, больше не является чем-то из ряда вон выходящим – подобный шаблон используется при разработке программного обеспечения для Интернета и на других платформах, таких как Angular 1.x и React.

Эта глава завершает знакомство с ядром Angular 2 и архитектурой приложений, использующих этот фреймворк. Теперь вы можете следовать выработанным стандартам при создании своих приложений, основанных на этом новом и интересном фреймворке.

В следующих главах мы сосредоточимся на вполне конкретных инструментах и модулях, используемых для решения повседневных задач, связанных с разработкой веб-проектов. Далее мы рассмотрим создание сетевых HTTP-клиентов с помощью Angular 2.

Глава 6

Асинхронная обработка данных в Angular 2

Подключение к службам данных и асинхронная обработка информации являются распространенной повседневной задачей разработчиков. Фреймворк Angular 2 предлагает для этих целей непревзойденный набор инструментов, вызывающий энтузиазм у разработчиков, когда дело доходит до получения, обработки и преобразования всех видов данных, возвращаемых службами.

Круг возможностей настолько широк, что потребовалась бы целая книга, чтобы описать все, что можно сделать для подключения к службе по протоколу HTTP или асинхронного извлечения информации из файловой системы. В этой книге они будут рассмотрены лишь поверхностно, но представленные здесь сведения о протоколе HTTP, сопутствующих классах и инструментах позволят вам подключать приложения к HTTP-службам, оставляя на ваше усмотрение все, что можно затем делать с ними.

В этой главе мы:

- рассмотрим различные способы асинхронной обработки данных;
- познакомимся с понятиями Наблюдаемые (Observables) и Наблюдатели (Observers);
- обсудим приемы функционального реактивного программирования и RxJS;
- исследуем класс `HTTP` и его интерфейс;

- разберем модуль `HTTP_PROVIDERS`;
- увидим все предыдущие пункты в действии на реальных примерах.

Способы асинхронной обработки информации

Извлечение информации из разного рода API представляет собой весьма распространенную в повседневной практике операцию. Получение информации через HTTP выполняется постоянно: при аутентификации пользователей, когда учетные данные отправляются в службу проверки подлинности, или при получении последних сообщений из популярного виджета Twitter. Современные мобильные устройства ввели уникальный способ использования удаленных служб, позволяющий откладывать посылку запросов и получение ответов до момента доступности мобильной связи. Интерактивности и доступности здесь придается важнейшее значение. Хотя современные интернет-соединения весьма быстры, между запросом и ответом всегда проходит какое-то время (время отклика), что требует создания механизмов обработки состояния в приложениях, функционирующих прозрачно для конечного пользователя.

Это относится не только к ситуациям, когда нужно получать информацию из внешнего ресурса. Иногда может понадобиться создать функционал, зависящий от времени как параметра чего-то и требующий применения шаблонов программирования для обработки отложенных изменений в состоянии приложения.

Во всех этих ситуациях используются такие шаблоны программирования, как, например, шаблон обратного вызова, при использовании которого функция, запускающая асинхронную операцию, ожидает вызова другой функции, определенной в ее сигнатуре и уведомляющей о завершении этой операции:

```
function notifyCompletion() {
  console.log('Our asynchronous operation has been completed');
}

function asynchronousOperation(callback) {
  setTimeout(callback, 5000);
}

asynchronousOperation(notifyCompletion);
```

Проблема этого шаблона в том, что с ростом приложения и увеличения количества вложенных обратных вызовов код может стать весьма запутанным и громоздким. Чтобы избежать этого, механизм Promise дает новый способ асинхронного управления данными, позволяющий писать более простой и надежный код, в котором различные асинхронные операции можно соединять на одном уровне и даже выделять и возвращать из других функций:

```
function notifyCompletion() {
  console.log('Our asynchronous operation has been completed');
}
function asynchronousOperation() {
  var promise = new Promise((resolve, reject) => {
    setTimeout(resolve, 5000);
  });
  return promise;
}

asynchronousOperation().then(notifyCompletion);
```

Этот пример получился более объемным, но функция asynchronousOperation в нем определенно имеет более наглядный и элегантный интерфейс.

Механизм Promise вихрем ворвался в мир программирования, и сейчас почти не осталось разработчиков, сомневающихся в его полезности. Тогда зачем нам другие парадигмы? Это связано с тем, что иногда может потребоваться получить ответ до завершения более сложного процесса или даже отменить этот процесс. Объекты Promise не позволяют этого, поскольку активируются в момент создания. Иными словами, объекты Promise не являются «ленивыми». С другой стороны, возможность разделения асинхронной операции на части после ее запуска, но до завершения, может быть очень полезной в определенных ситуациях. Объекты Promise позволяют только принять или отклонить асинхронную операцию, но иногда нужно отменить ее прежде, чем она завершится. Кроме того, объекты Promise действуют как однократные операции. После их разрешения нельзя получить никакой дополнительной информации или уведомления об изменении состояния, если только не повторить все сначала. Кроме того, иногда возникает необходимость в более активной реализации асинхронной обработки данных. В этом могут помочь наблюдаемые объекты.

Наблюдаемые объекты

Наблюдаемый объект (Observable) обычно выступает источником асинхронного события, сообщающего другому элементу, называемому наблюдателем (Observer), об изменении состояния. Для этого наблюдаемый объект реализует все механизмы, которые создают и рассылают такие асинхронные события, и он может быть запущен или остановлен в любое время, независимо от того, были переданы ожидаемые данные или нет.

Этот шаблон поддерживает конкурирующие операции и более продвинутую логику, поскольку наблюдатели, подписанные на асинхронные события, будут реагировать на изменение состояния наблюдаемого объекта.

Подписчики, являющиеся упомянутыми ранее наблюдателями, следят за наблюдаемым объектом, пока тот не будет удален, если такое вообще произойдет. При этом информация будет обновляться во всем приложении, без необходимости запускать какие-либо процедуры.

Для большей наглядности рассмотрим реальный пример. Возьмем за основу пример, демонстрирующий асинхронные операции с использованием `Promise`, и заменим команду `setTimeout` командой `setInterval`:

```
function notifyCompletion() {
  console.log('Our asynchronous operation has been completed');
}

function asynchronousOperation() {
  var promise = new Promise((resolve, reject) => {
    setInterval(resolve, 2000);
  });
  return promise;
}

asynchronousOperation().then(notifyCompletion);
```

Скопируйте и вставьте предыдущий фрагмент в окно консоли браузера и посмотрите, что при этом произойдет. Текст **«Our asynchronous operation has been completed»** появится в консоли только один раз, по истечении 2 секунд, и не будет выведен повторно. Объект `Promise` был разрешен и в тот же момент остановлен.

А теперь перейдите в браузере к одной из песочниц для кода на JavaScript в Интернете, например на сайте JSBIN (https://jsbin.

com/), и создайте новый фрагмент кода, используя только вкладки **JavaScript** и **Console**. Добавьте библиотеку **RxJS library**, выбрав ее в раскрывающемся списке **Add library** (она необходима для создания наблюдаемых объектов; не волнуйтесь, мы рассмотрим ее далее в этой главе), и вставьте следующий фрагмент кода:

```
var observable = Rx.Observable.create(observer => {
  setInterval(() => {
    observer.onNext('My async operation');
  }, 2000);
});

observable.subscribe(response => console.log(response));
```

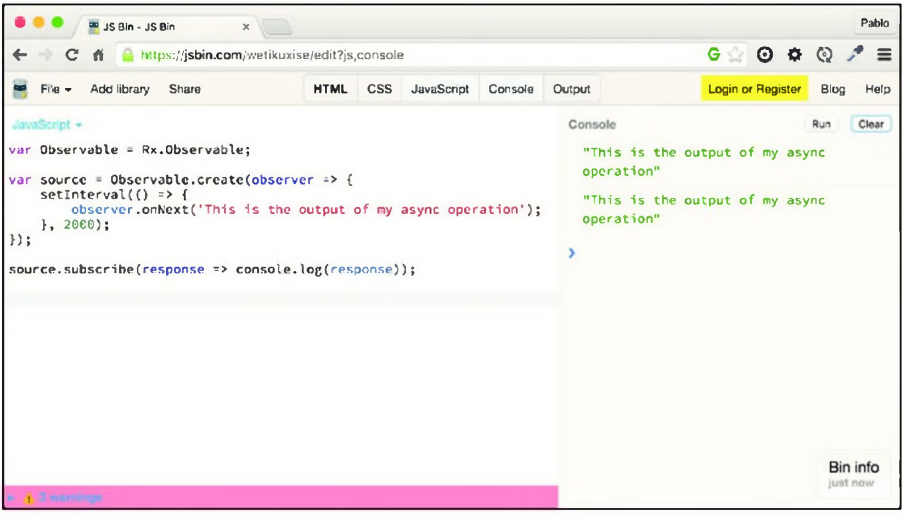


Рис. 6.1

Запустите его и дождитесь вывода сообщения в правой панели. Через 2 секунды будет выведено то же сообщение, что и выше, а затем еще одно, и еще, и еще... В этом простом примере мы создали наблюдаемый объект, подписались на его события и организовали вывод в консоль передаваемых им данных (в данном примере это простое сообщение).

Наблюдаемый объект возвращает поток событий, и подписчики, оперативно получая уведомления об этих событиях, соответственно реагируют. Именно на этом основана магия наблюдаемых объектов. Наблюдаемые объекты не просто выполняют асинхронную операцию и затем пропадают (хотя в них можно настроить и такой порядок дей-

ствий), а производят непрерывный поток событий, на которые можно подписывать абонентов.

Если закомментировать последнюю строку кода, ничего не произойдет. В панель **Console** ничего не будет выводиться, вся магия начнется с момента, когда осуществляется подписка.

Но и это еще не все. Этот поток можно подвергнуть дополнительной обработке, прежде чем он достигнет подписанных на него наблюдателей. Подобно тому, как можно взять объект коллекции, например массив, и применить к нему функциональные методы, такие как `map()` или `filter()`, чтобы преобразовать массив и изменить его элементы, с потоком событий можно сделать то же самое. Это называется реактивным функциональным программированием, и фреймворк Angular 2 применяет значительную часть этой парадигмы для асинхронной обработки информации.

Реактивное функциональное программирование в Angular 2

Шаблон наблюдения (Observable), по сути, является основой того, что носит название реактивное функциональное программирование. Большинство базовых реализаций сценариев реактивной функциональности опирается на несколько концепций, с которыми следует познакомиться:

- наблюдаемый объект (Observable);
- наблюдатель (Observer);
- временная шкала;
- поток событий, характеризующийся тем же поведением, что и коллекция объектов;
- набор компонуемых операторов, называемый также *реактивным расширением.*

Сложно? Не совсем. Поверьте на слово, что код, который был приведен выше, гораздо сложнее. Самое сложное – поменять образ мышления, освоить умение мыслить в реактивном стиле, и в этом состоит основная цель данного раздела.

Проще говоря, реактивное программирование заключается в подписке на асинхронные события и преобразованиях потоков наблюдаемых событий. Могу представить себе ваше лицо, похожее сейчас на лицо играющего в покер, так что перейдем к рассмотрению наглядного примера.

Представьте, как действует, например, такое устройство, как клавиатура. На клавиатуре имеются клавиши, которые нажимает пользователь. Нажатие любой клавиши вызывает событие нажатия клавиши. Событие нажатия клавиши характеризуется набором метаданных, включающим в числе прочих числовой код нажатой в данный момент клавиши. По мере того как пользователь продолжает нажимать клавиши, возникает все больше событий `keyUp`, и они располагаются вдоль воображаемой временной шкалы. Поздравляю! Здесь продемонстрирован порядок наблюдения в действии: клавиатура берет на себя роль наблюдаемого и порождает последовательность событий, информирующих о нажатиях клавиш, независимо от того, что произойдет дальше. Каждое событие не зависит от остальных, и все они следуют друг за другом во времени. Здесь было упомянуто следование событий, одного за другим. Если взять группу элементов заданного типа (или даже разных типов) и обернуть их в объект (которым может быть временная шкала), то разве это не коллекция? Кроме того, эта коллекция учитывает время. Так что же это, если не поток? Если вы поняли суть, примите поздравления! Теперь у вас имеется поток событий, генерируемый наблюдаемым объектом для наблюдателей, как показано на рис. 6.2.

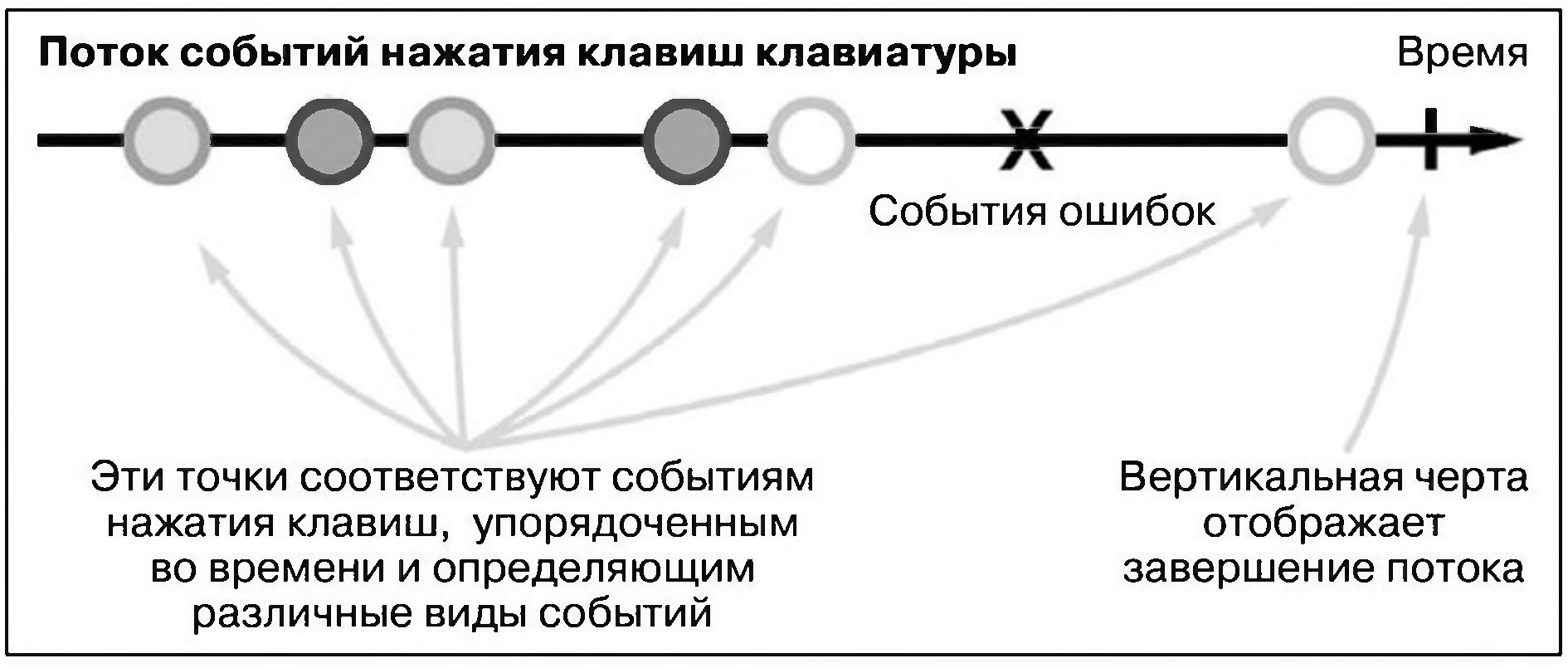


Рис. 6.2

Поток событий нажатия клавиш выглядит отлично, но содержит уведомления о нажатии всех клавиш. А что делать, если нужно подписаться на подмножество потока, содержащее только события нажатия клавиш управления курсором? Допустим, разрабатывается простая игра, и игрок управляет персонажем с помощью курсорных клавиш. События, вызванные нажатием других клавиш, здесь совершенно

бесполезны. Тогда, в идеале, поток должен содержать только события нажатия клавиш управления курсором. Здесь вступает в игру функциональная часть реактивной парадигмы.

Представим, что имеется возможность перехватить поток событий и отфильтровать его так, чтобы он содержал только события нажатия курсорных клавиш, игнорируя все другие события. Кроме того, следует замедлить поток, разрешив включение в него событий нажатия клавиш не чаще, чем один раз в 500 миллисекунд, чтобы не вызвать переполнения у наблюдателей на другом конце провода. Теперь имеется новый поток событий, являющийся побочным продуктом исходного и представляющий собой подмножество данных для решения конкретной задачи, как показано на рис. 6.3.

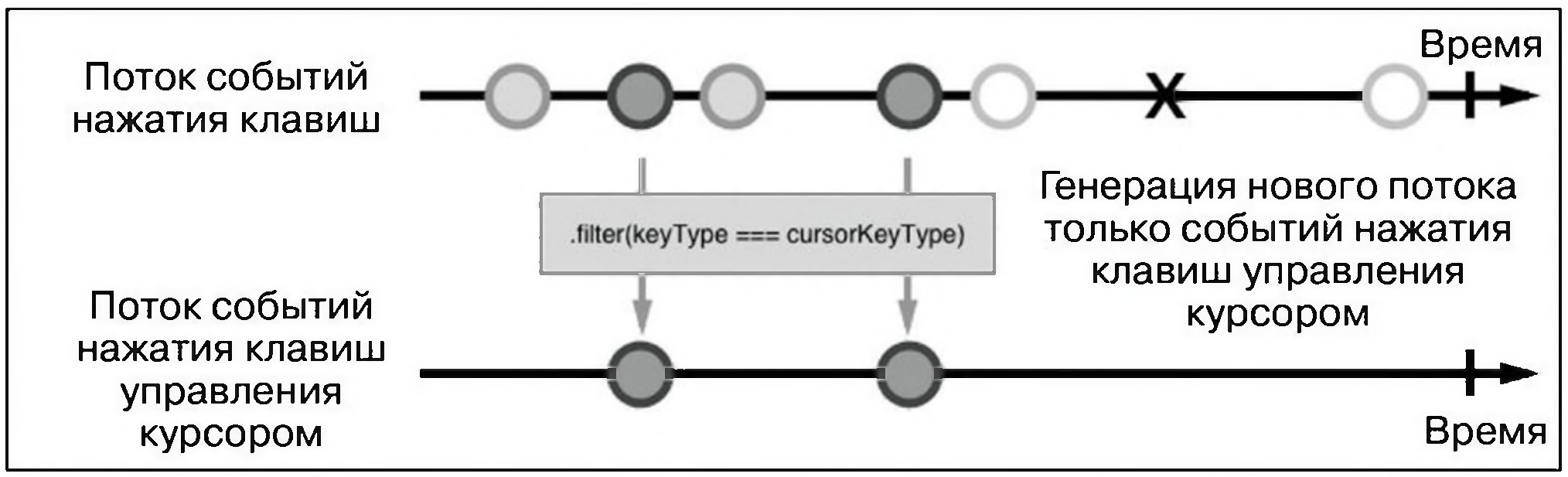


Рис. 6.3

Подписавшись на этот поток, можно реализовать выполнение операций при нажатии любой из клавиш управления курсором и применить логику разрабатываемой игры. Также к этому потоку можно подключить несколько подписчиков или применить дальнейшие преобразования, с использованием других реактивных расширений, чтобы создать совершенно новые потоки, на которые могут подписаться другие абоненты, реализующие другую бизнес-логику, никак не связанную с выполняемой оригинальным подписчиком.

Эта логика может относиться к сфере компонентов для обработки событий взаимодействия, асинхронного поведения или (как описано в этой главе) извлечения и обработки информации, предоставляемой службами или хранилищами данных. При реализации этой возможности фреймворк Angular 2 полагается на библиотеку RxJS, включенную в число его основных зависимостей. Библиотека RxJS содержит все необходимое для реализации описанных выше функций, осуществляющих асинхронные операции.

Библиотека RxJS

Как уже упоминалось ранее, фреймворк Angular 2 поставляется с учетом зависимости от библиотеки RxJS, версии библиотеки `ReactiveX` для JavaScript, которая позволяет создавать наблюдателей и наблюдаемые объекты для широкого диапазона потребностей, таких как работа с интерактивными событиями, объектами `Promise` и функциями обратного вызова, что составляет далеко не полный список. Поддержка реактивного программирования не стремится заменить асинхронные шаблоны, такие как `promise` или обратные вызовы, но может использоваться в дополнение к ним, а также для создания механизмов наблюдения.

Библиотека RxJS поставляется со встроенной поддержкой широкого спектра компонуемых операторов для преобразования, фильтрации и комбинирования потока событий. Ее интерфейс включает удобные методы для подписки наблюдателей на потоки, что позволяет сценариям и компонентам реагировать соответствующим образом на изменение состояния или интерактивный ввод. Поскольку этот программный интерфейс огромен и детальное его рассмотрение выходит за рамки книги, будет описана лишь небольшая его часть, что позволит разобраться в работе с HTTP-соединениями с помощью Angular 2.

Перед тем как углубиться в поддержку HTTP, предоставляемую Angular 2, рассмотрим простой пример потока наблюдаемых событий, который будет подвергнут преобразованию с помощью реактивных расширений и передан подписавшимся на него наблюдателям. Выберем для примера задачу, описанную в предыдущем разделе.

Она предусматривает взаимодействие пользователя с приложением посредством клавиатуры, которое сводится к нажатию клавиш с течением времени и, следовательно, может рассматриваться как поток событий. Вернитесь на сайт JSBIN, удалите содержимое в панели **JavaScript**, а затем введите следующий фрагмент:

```
var keyboardStream = Rx.Observable
  .fromEvent(document, 'keyup')
  .map(x => x.which);
```

Предыдущий код интуитивно понятен. Здесь использованы класс `Rx.Observable` и его метод `fromEvent`, определяющий источник, который генерирует поток событий `keyup` в рамках объекта `document`. Генерируемые события представляют собой слишком сложные объекты, поэтому путем отображения потока событий в новый поток произво-

дится их упрощение до объектов, содержащих только коды нажатых клавиш. Метод `map` – это реактивное расширение, которое ведет себя так же, как функциональный метод `map` в JavaScript. Именно поэтому этот стиль программирования называют реактивным функциональным программированием.

Итак, теперь поток событий содержит лишь числовые коды клавиш, но нас интересуют лишь события нажатия клавиш управления курсором. Чтобы отфильтровать ненужные события, из уже имеющегося потока можно создать новый, применив другие реактивные расширения. Профильтруем поток с помощью `keyboardStream` и вернем только события, связанные с курсорными клавишами, и дополнительно преобразуем эти события в соответствующий им текст, для большей наглядности. Добавьте следующий фрагмент непосредственно за предыдущим:

```
var cursorMovesStream = keyboardStream.filter(x => {
    return x > 36 && x < 41;
  })
  .map(x => {
    var direction; switch(x) {
      case 37:
        direction = 'left'; break;
      case 38:
        direction = 'up'; break;
      case 39:
        direction = 'right'; break;
      default:
        direction = 'down';
    }
  return direction;
});
```

Тот же результат можно получить, объединив методы `filter` и `map` объекта `keyboardStream` в одну цепочку с вызовом метода `Observable`, а затем подписавшись на результат, но часто лучше разделять такие задачи. При организации кода, как в примере выше, сохраняется общий поток событий клавиатуры, который в дальнейшем можно использовать в других целях. Благодаря этому приложение можно масштабировать, сведя к минимуму объем необходимого кода.

Теперь, после упоминания о подписке, осуществим подписку на поток событий нажатия клавиш управления курсором и с ее помощью выведем команды управления в консоль. Добавьте следующую ин-

струкцию в конец сценария, затем очистите панель **Console** и щелкните на вкладке **Output**, чтобы получить доступ к документу:

```
cursorMovesStream.subscribe(e => console.log(e));
```

Щелкните в любом месте на панели **Output** для передачи ей фокуса ввода и нажмите несколько случайных клавиш и клавиши управления курсором, результат должен быть похож на показанный на рис. 6.4.

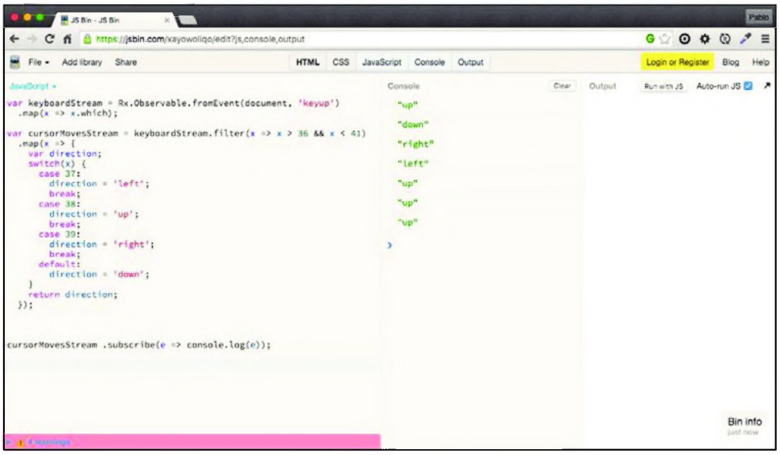


Рис. 6.4

Вам, вероятно, интересно, как применить этот шаблон для выполнения асинхронных операций, таких как получение информации от HTTP-служб. Мы привыкли считать, что для отправки асинхронных запросов следует использовать службы AJAX, а обработку ответов делегировать функциям обратного вызова или пропускать их через объекты `promise`. Теперь мы пойдем другим путем и реализуем обработку запросов, возвращая наблюдаемый объект. Наблюдаемый объект будет генерировать возвращение сервера в виде события в контексте потока, который мы пропустим через реактивные расширения для преобразования ответа в нужную форму.

Введение в HTTP API

Класс `Http` имеет мощный программный интерфейс, абстрагирующий все операции, необходимые для обработки асинхронных соединений посредством различных HTTP-методов, предоставляя простой и удобный способ обработки ответов. При его реализации особое внимание уделялось удобству разработки решений, использующих этот класс для подключения к службам и источникам данных.

Экземпляры класса `Http` (реализованного в виде внедряемого ресурса, что дает возможность использовать его в конструкторах классов посредством объявления в качестве поставщика зависимости) экспортируют метод подключения `request()` для выполнения любого типа `http`-соединения. Команда Angular 2 создала специальный сокращенный синтаксис для наиболее популярных операций запросов, таких как `GET`, `POST`, `PUT` и других HTTP-команд, упрощающий создание асинхронных `Http`-запросов, как показано ниже:

```
var requestOptions = new RequestOptions({
  method: RequestMethod.Get,
  url: '/my-api/my-data-store.json'
});
var request = new Request(requestOptions);
var myHttpRequest: Observable<Response> = http.request(request);
```

Кроме того, все это можно свести в единственную строку кода:

```
var myHttpRequest: Observable<Response> =
  http.get('/my-api/my-data-store.json');
```

Как видите, методы подключения класса `Http` возвращают наблюдаемый поток экземпляров объекта `Response`. Это позволяет применить дополнительную обработку к полученному результату и подписать наблюдателей, которые, соответственно, будут обрабатывать возвращаемую информацию столько раз, сколько требуется:

```
myHttpRequest.map(response: Response => response.json())
  .subscribe(data => console.log(data));
```

В приведенном примере поток наблюдаемых событий (которые, по сути, являются экземплярами класса `Response`, представляемого фреймворком Angular 2) отображается в новую коллекцию событий, состоящую из JSON-экземпляров ответов. Это достигается с помощью метода `json()` класса `Response`, который будет рассмотрен ниже в этой главе. Затем осуществляются подписка на получившийся по-

ток и вывод HTTP-ответов в панель **Console** после получения наблюдателем уведомления о данных.

Таким способом можно выполнить запрос `Http` столько раз, сколько потребуется, остальная часть механизма отреагирует соответствующим образом. Можно даже объединить поток событий, возвращаемый вызовом `Http`, с другими связанными вызовами и создать более сложные наблюдаемые потоки данных. Возможности здесь бесконечны.

В каких случаях используются классы Request и RequestOptionsArgs

Классы `Request` и `RequestOptions` уже упоминались при знакомстве с классом `Http`. Обычно нет необходимости использовать эти низкоуровневые классы, главным образом потому, что сокращенные методы класса `Http` позволяют не указывать команды HTTP (`GET`, `POST` и т. д.) и адрес URL. Тем не менее иногда нужно вставить в запрос специальные HTTP-заголовки или добавить строковые параметры. Поэтому использование этих классов очень удобно в определенных случаях. Рассмотрим случай, когда требуется добавить в каждый из запросов токен аутентификации с целью предотвращения несанкционированного чтения данных посредством программного интерфейса конечной точки.

Следующий пример читает токен аутентификации, а затем добавляет его в виде заголовка в запрос к службе данных. В примере объект хэша вводится прямо в конструкторе `Request`, что означает пропуск этапа создания экземпляра объекта `RequestOptions`. Angular 2 предоставляет класс-оболочку для определения пользовательских заголовков, которая и будет использована в данном случае. Предположим, например, что служба требует, чтобы все запросы включали специальный заголовок `Authorization` со значением `authToken`, полученным при входе в систему и сохраненым в локальном хранилище браузера:

```
var authToken = window.localStorage.getItem('auth_token');
var headers = new Headers();
headers.append('Authorization', `Token ${authToken}`);

var request = new Request({
  method: RequestMethod.Get,
  url: '/my-api/my-secured-data-store.json',
  headers: headers
});

var authRequest: Observable<Response> = http.request(request);
```

Отмечу еще раз, что, помимо этой ситуации, редко бывает нужно создавать пользовательские конфигурации запросов, если только не требуется делегировать создание таких конфигураций классу или методу фабрики и повторно использовать одну и ту же `Http`-оболочку. Фреймворк Angular 2 обладает достаточной гибкостью, чтобы обеспечить любой уровень абстрагирования в приложении.

Объект Response

Как уже упоминалось выше, HTTP-запросы, выполняемые классом `Http`, возвращают поток экземпляров класса `Response`. По аналогии с объектом `Request` вам редко придется напрямую использовать экземпляры этого класса. Тем не менее знание интерфейса класса `Response` весьма полезно для понимания состояния запроса, обработки ошибок подключения и правильного извлечения сведений из потока.

В первом примере мы отобразили содержимое потока в поток JSON-объектов с помощью метода `json()`. Этот метод анализирует тело ответа как JSON-объект, вызывая исключение, если тело не может быть так интерпретировано. Кроме этого метода, класс `Response` экспортирует метод `text()`, который анализирует и возвращает тело ответа в виде обычной строки.

Для иллюстрации предположим, что имеется компонент со строковым полем `bio`, значение которого выводится в шаблоне компонента. Значение свойства `bio` может быть получено, например, из REST API. Подобную функциональность можно реализовать всего в нескольких строках кода:

```
http.get('/api/bio')
  .map(res: Response => res.txt)
  .subscribe(bio => this.bio = bio);
```

Объект `Response` экспортирует также другие вспомогательные методы и несколько интересных свойств, например числовое свойство `status`, которое содержит код состояния, возвращенный сервером. Числовые свойства `bytesLoaded` и `totalBytes` могут пригодиться для создания уведомлений о ходе выполнения процесса. Особого упоминания заслуживает также свойство `headers`, которое содержит объект, базирующийся на классе `Headers` (https://fetch.spec.whatwg.org/#headers-class) прикладного программного интерфейса Fetch.

Описание всех методов и свойств класса `Response`, естественно, выходит за рамки этой книги, поскольку они используются достаточ-

но редко. Более подробные сведения по этому вопросу можно найти в официальной документации (https://angular.io/docs).

Обработка ошибок при выполнении Http-запросов

Обработка ошибок, возникающих при выполнении запросов, основывается на сведениях, возвращаемых в объекте `Response`, и довольно проста. Нужно лишь проверить логическое свойство, которое получает значение `false`, если код состояния HTTP не находится в диапазоне значений 2xx, что явно указывает на ошибку. Дополнительно можно проанализировать код ошибки в свойстве `status` или содержимое свойства `type`, которое может принимать значения `basic`, `cors`, `default`, `error` или `opaque`. Свойства `headers` и `statusText` объекта `Response` содержат более подробную информацию о причине ошибки.

Но в общем случае нет смысла просматривать эти свойства в каждом поступающем сообщении. Для перехвата ошибок фреймворк Angular 2 предоставляет специальный оператор. Добавив его в сигнатуру объекта `Response`, можно просмотреть значения описанных выше свойств:

```
http.get('/api/bio')
  .map(res: Response => res.txt)
  .subscribe(bio => this.bio = bio)
  .catch(error: Response => console.error(error));
```

Обычно желательно проанализировать больше сведений, а не только значение свойства `error`, предусмотрев более солидную систему регистрации ошибок, чем та, что реализована выше.

Внедрение класса Http и псевдоним HTTP_PROVIDERS

Класс `Http` можно внедрять в пользовательские компоненты и классы с помощью уникальной системы внедрения зависимостей Angular. Поэтому, если потребуется реализовать HTTP-вызовы, следует импортировать этот класс и указать его как зависимость в списке поставщиков компонент или директив:

```
import { Component } from '@angular/core';
import { Http } from '@angular/http';

@Component({ selector: 'bio', providers: [Http],
  template: '<div>{{bio}}</div>'
})
```

```
class Biography {
  bio: string;
  constructor(http: Http) {
    http.get('/api/bio')
    .map((res: Response) => res.text())
    .subscribe((bio: string) => this.bio = bio);
  }
}
```

Приведенный выше код является продолжением примера `bio` из предыдущего раздела. Обратите внимание, как здесь импортируется тип `Http` и внедряется в качестве зависимости посредством свойства-коллекции `providers` декоратора `Component`.

Часто требуется выполнить несколько HTTP-вызовов в различных частях приложения, поэтому обычно рекомендуется включать класс `Http` в корневой инструмент внедрения зависимостей, а не в инструмент внедрения зависимостей отдельного компонента. С учетом того, что классу `Http` могут потребоваться другие поставщики, такие как класс `RequestOptions`, фреймворк Angular 2 предоставляет псевдоним `HTTP_PROVIDERS`, обертывающий набор внедряемых типов.

Для внедрения зависимостей, необходимых для выполнения HTTP-запросов, вместо класса `Http` рекомендуется использовать этот набор. Для удобства добавьте его в свойство `providers` корневого компонента, чтобы сделать этого поставщика доступным для всех дочерних компонентов. В примере приложения Pomodoro нужно импортировать его в блоке кода `AppComponent` и здесь же добавить в свойство `providers`:

APP/APP.COMPONENT.TS

```
import { Component } from '@angular/http';
import { TIMER_DIRECTIVES } from './timer/timer';
import { TASKS_DIRECTIVES } from './tasks/tasks';
import { SHARED_PROVIDERS } from './shared/shared';
import { HTTP_PROVIDERS } from '@angular/http';

@Component({
  selector: 'pomodoro-app',
  directives: [TIMER_DIRECTIVES, TASKS_DIRECTIVES],
  providers: [SHARED_PROVIDERS, HTTP_PROVIDERS],
  ...
})
export default class AppComponent {}
```

Пример: отслеживание данных по HTTP

В предыдущей главе весь код приложения был разделен на файлы моделей, служб, фильтров, директив и компонентов. Одной из служб был класс `TaskService`, который является основой всего приложения, поскольку предоставляет данные для построения списка заданий и других связанных с ним компонентов.

Класс `TaskService` содержит исходную информацию. В реальной ситуации эту информацию придется извлекать из серверной библиотеки или службы. Давайте реализуем этот сценарий.

Сначала удалим сведения о заданиях из класса `TaskService` и поместим их в файл JSON. Для этого создадим новый файл JSON в общей папке и заполним его сведениями о заданиях, перенеся их из файла `TaskService.ts` и преобразовав в формат JSON:

APP/SHARED/DATA/RAW-TASKS.JSON

```
[{
  "name": "Code an HTML Table",
  "deadline": "Jun 23 2015",
  "pomodorosRequired": 1
}, {
  "name": "Sketch a wireframe for the new homepage",
  "deadline": "Jun 24 2016",
  "pomodorosRequired": 2
}, {
  "name": "Style table with Bootstrap styles",
  "deadline": "Jun 25 2016",
  "pomodorosRequired": 1
}, {
  "name": "Reinforce SEO with custom sitemap.xml",
  "deadline": "Jun 26 2016",
  "pomodorosRequired": 3
}]
```

Теперь, когда данные помещены в отдельный файл, их можно получить с помощью клиентского класса `TaskService`, как если бы он был настоящей серверной службой. Но для этого понадобится внести соответствующие изменения в файл `main.ts`. Дело в том, что, несмотря на наличие библиотеки `RxJS`, установленной вместе с остальными зависимостями Angular 2, функциональные реактивные операторы, такие как `map()`, остаются недоступными в приложении. Чтобы получить возможность пользоваться ими, их нужно импортировать.

Импортировать можно сразу все, добавив следующую строку где-то в блоке инициализации приложения, например на стадии начальной загрузки в файле `main.ts`:

```
import 'rxjs/Rx';
```

Однако если импортировать все операторы, большая часть из которых вообще не будет использоваться, это приведет к напрасной трате пропускной способности и ресурсов. Поэтому предпочтительнее импортировать только то, что действительно необходимо, поэтому добавим следующие операторы импорта в начало файла `main.ts`:

APP/MAIN.TS

```
import 'rxjs/add/operator/map';
import { bootstrap } from '@angular/platform-browser-dynamic';
import AppComponent from './app.component';

bootstrap(AppComponent, []);
```

Когда реактивный оператор импортируется таким способом, он автоматически добавляется в прототип `Observable` и становится доступен для использования во всем приложении.

После настройки зависимостей можно приступать к внесению изменений в файл `TaskService.ts`. Откройте его и обновите блок операторов импорта:

APP/SHARED/SERVICES/TASK.SERVICE.TS

```
import { Injectable } from '@angular/code';
import { Task } from '../shared';
import { Http, Response } from '@angular/http';
import { Observable } from 'rxjs/Observable';
```

В первую очередь мы импортировали символы `Http` и `Response`, чтобы в дальнейшем аннотировать ими свои объекты. В любом случае, имейте в виду, что псевдоним `HTTP_PROVIDERS` уже был внедрен в корневой компонент верхнего уровня. Символ `Observable` импортируется из библиотеки `RxJS`, чтобы можно было аннотировать возвращаемые типы асинхронных запросов `Http`.

А теперь заменим существующую реализацию приведенной ниже. По сути, модуль `TaskService` превратился в службу с состоянием, которая экспортирует свойство `taskStore`, откуда можно извлечь набор заданий. Она также имеет свойство `Observable` для подписки на уведомления о создании новых заданий.

Конструктор теперь внедряет зависимость `Http` и привязывает ее к закрытому полю `http`:

APP/SHARED/SERVICES/TASK.SERVICE.TS

```
@Injectable()
export default class TaskService {
taskStore: Task[] = [];
taskFeed: Observable<Task>;
private taskObserver: any;
private dataUrl = '/app/shared/data/raw-tasks.json'

constructor(private http: Http) {
  this.taskFeed = new Observable(observer => {
    this.taskObserver = observer;
  });
  this.fetchTasks();
}

private fetchTasks(): void {
  this.http.get(this.dataUrl)
    .map(response => response.json())
    .map(stream => stream.map(res => {
      return {
        name: res.name,
        deadline: new Date(res.deadline),
        pomodorosRequired: res.pomodorosRequired,
        queued: res.queued
      }
  }))
  .subscribe(
    tasks => {
      this.taskStore = tasks;
      tasks.forEach(task => this.taskObserver.next(task))
    },
    error => console.log(error)
  );
}
```

Уделим немного времени новой реализации службы. В ней сохранилось свойство `taskStore`, заполняемое после извлечения всего списка заданий, получаемого при создании экземпляра службы вызовом закрытого метода `fetchTasks()` в конструкторе. А также добавились поля для наблюдаемого объекта и наблюдателя.

Опишем их роли подробно. Конструктор инициализирует экземпляр наблюдаемого объекта и присваивает встроенного наблюдателя

свойству для наблюдения. Благодаря этому, когда потребуется известить подписчиков о появлении нового задания, мы сможем просто вызвать метод `next()` элемента наблюдателя. Следовательно, служба заданий, как синглтон, экспортирует хранилище данных, заполняемое в момент ее создания, делая данные доступными всем нуждающимся в них компонентам.

С другой стороны, внешние абоненты могут подписаться на свойство `taskFeed` и оперативно получать уведомления при добавлении новых заданий. Можно создать клиента веб-сокетов и использовать интерфейс наблюдателя для передачи новых заданий через наблюдаемую последовательность при их создании на стороне сервера. Подписанные на наблюдаемую последовательность компоненты будут уведомляться об изменениях, которые автоматически будут отражаться на их состоянии без привлечения дополнительной логики.

Как видите, этот шаблон обработки данных простирается гораздо дальше простого асинхронного приема потока данных, выполняемого привычными шаблонами `promise` и обратными вызовами, упрощая обновление данных и позволяя более эффективно управлять информацией в автономном режиме.

Закончим эту реализацию, внеся изменения в модуль `TaskComponent`. Поскольку используется тот же самый программный интерфейс, что и ранее, изменения заключаются лишь в настройке метода-обработчика `ngOnInit`:

APP/TASKS/TASKS.COMPONENT.TS

```
ngOnInit(): void {
  this.updateQueuedPomodoros();
  this.taskService.taskFeed.subscribe(newTask => {
    this.tasks.push(newTask);
    this.updateQueuedPomodoros();
  });
}
```

Для активации наблюдаемых объектов необходимо подписаться на получение заданий. Они не будут инициализированы, пока какой-либо клиент не присоединится к ним, и это объясняет необходимость присутствия основного наблюдателя для объекта `taskFeed` в службе `TaskService`, осуществляющего внутреннюю подписку на соединение `http.get()`. Отсутствие такой подписки вызовет исключение. Кроме того, эти изменения гарантируют сохранение списка заданий до

момента добавления нового задания в общее хранилище, избавляя от необходимости повторного заполнения свойства `tasks`. Метод `updateQueuedPomodoros()` используется в обработчике для автоматического включения новых заданий в очередь.

Запустив приложение, можно убедиться в безупречном отображении таблицы заданий.

Добавление новых заданий

К сожалению, сейчас служба работает односторонне: мы можем получать данные из файла JSON, но не можем добавлять новые задания. В идеале следует предусмотреть возможность добавления собственных заданий по запросу с реагированием всей системы на такие изменения, поэтому модифицируем реализацию класса `TaskService` и добавим метод вставки и добавления заданий:

APP/SHARED/SERVICES/TASK.SERVICE.TS

```
addTask(task: Task): void {
  this.taskObserver.next(task);
}
```

Этот метод опирается на уже подготовленный реактивный интерфейс службы. Добавление нового задания превратится в новое событие, опубликуется в потоке, и любой активный компонент, принимающий список заданий и подписанный на его изменения, оперативно получит уведомление о вновь созданном задании и сможет соответствующим образом обновить свое состояние.

Попробуйте сами!



Необходимо отметить, что запрос на добавление нового элемента должен выполняться как запрос `POST`, и все предыдущие операции должны выполняться после вызова `Http.post()`, например:

```
const body = JSON.stringify(task); const headers = new Headers();
headers.append('Content-Type', 'application/json');
```

```
addTask(task: Task): void {
    this.http.post(this.dataUrl, body, headers)
      .map(response => response.json())
      .subscribe((task: Task) => this.taskObserver.next(task);
        }
      );
  }
```

Поскольку обзор серверной реализации выходит за рамки этой книги, предлагаю поэкспериментировать с модулем `Http` и сравнить его с прикладным программным интерфейсом RESTful.

Итоги

Как было упомянуто в начале главы, подробное рассмотрение всех возможностей Angular 2 при работе с HTTP-соединениями невозможно уместить в одну главу, тем не менее здесь были рассмотрены практически все необходимые инструменты и классы.

Все прочее ограничивается лишь вашей фантазией. Попробуйте, например, применить полученные знания на практике и создать новый Twitter-клиент, новостной виджет или движок блога, используя набор компонентов по своему выбору. Возможности безграничны, и вы можете использовать любые подходы, от шаблона `Promise` до наблюдения. Воспользуйтесь возможностями реактивных расширений и крошечного, но мощного класса `Http`.

Как уже было отмечено, ограничений не существует. Но предстоит еще долгий и увлекательный путь. После знакомства с приемами асинхронной обработки данных в компонентах рассмотрим способы улучшения взаимодействия с пользователем, основанные на маршрутизации. Этому будет посвящена следующая глава.

Глава 7

Маршрутизация в Angular 2

В предыдущих главах была проделана большая работа по разделению областей ответственности в приложении и добавлению уровней абстракции, направленная на облегчение сопровождения приложения Pomodoro. Однако мы оставили без внимания внешнюю сторону и удобство восприятия пользователем.

На данный момент пользовательский интерфейс загромождён компонентами, переполняющими единственную страницу, и нужно обеспечить навигацию и интуитивный способ изменения состояния приложения.

Здесь особое значение приобретает маршрутизация, дающая возможность создать навигационный сценарий приложения, что позволит разбить различные области интересов на несколько страниц, соединенных ссылками и URL-адресами.

Но приложение содержит лишь набор компонентов, как же организовать схему переходов между ними? Маршрутизатор Angular 2 изначально предназначен для работы с компонентами. Здесь будет описан процесс создания пользовательских ссылок и обеспечения реакции компонентов на переход к следующий странице.

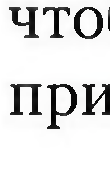
В этой главе мы:

- узнаем, как определять маршруты для включения и выключения компонентов, а также их перенаправления;
- научимся активировать маршруты и загружать компоненты в представления, в зависимости от запрошенного маршрута;
- освоим передачу параметров компонентам непосредственно из маршрута;

- рассмотрим разные обработчики событий жизненного цикла компонентов, основанные на этапах маршрутизации;
- познакомимся с разными способами представления URL-адресов.

Добавление поддержки маршрутизатора Angular 2

По аналогии с директивами и поставщиками зависимостей Http, все типы и псевдонимы, необходимые для реализации поддержки маршрутизации в приложениях, включены в специальный баррель. Этот баррель уже был установлен и настроен в *главе 1 «Создание самого первого компонента в Angular 2»*, но имеются еще два барреля, относящиеся к маршрутизации, которые были установлены: `@angular/router` и `@angular/router-deprecated`. Их присутствие объясняется тем, что команда Angular при переходе от бета-версии к версии релиз-кандидат добавила усовершенствованный механизм маршрутизации. Новый механизм, призванный заменить прежний интерфейс маршрутизации и реализованный еще в альфа-версии, вносит критические изменения, ломающие обратную совместимость. Чтобы приложения, основанные на предыдущем маршрутизаторе, можно было без больших проблем обновить для работы с новой версией Angular 2, команда Angular сделала доступной текущую версию маршрутизатора в виде барреля `@angular/router-deprecated`. Это и вызвало необходимость установки и настройки двух пакетов маршрутизации.



На момент написания этой книги новый маршрутизатор Angular 2 все еще находится в достаточно сыром состоянии и не имеет поддержки нескольких функций, которые необходимы в обычной повседневной работе. Поэтому в этой главе разработка приложений будет основываться на устаревшем маршрутизаторе, но по мере возможности будут упоминаться различия старого и нового маршрутизаторов. Однако различия минимальны, а навыки использования устаревшего маршрутизатора позволят значительно ускорить процесс освоения нового маршрутизатора, после того как его разработка будет завершена. Последнюю версию кода можно найти в репозитории книги.

Также необходимо сообщить фреймворку Angular базовый путь, чтобы он мог должным образом строить и распознавать URL-адреса при просмотре сайта пользователем, как будет продемонстрировано

в следующем разделе. Первая наша задача: вставить оператор base href в элемент <HEAD>. Добавьте следующую строку кода в конец тега <head>:

INDEX.HTML

```
<base href="/">
```

Тег base сообщает браузеру путь, по которому он должен следовать при попытке загрузить внешние ресурсы, такие как медиа- и CSS-файлы, после углубления в иерархию URL-адресов.

А теперь можно начинать экспериментировать с возможностями библиотеки маршрутизатора. Но прежде нужно сообщить механизму внедрения зависимостей, как создавать экземпляры типов, которые понадобятся в дальнейшем для реализации маршрутизации в компонентах. Все нужные нам поставщики объединены в псевдониме ROUTER_PROVIDERS. По аналогии с псевдонимом HTTP_PROVIDERS, его следует объявить в свойстве providers корневого компонента верхнего уровня, чтобы сделать доступным для внедрения зависимостей во всех дочерних компонентах.

Откройте модуль корневого компонента и добавьте следующий оператор import в блок импорта. Затем добавьте псевдоним в свойство providers декоратора компонента:

APP/APP.COMPONENT.TS

```
...
import { SHARED_PROVIDERS } from './shared/shared';
import { HTTP_PROVIDERS } from '@angular/http';
import { ROUTER_PROVIDERS } from '@angular/router-deprecated';

@Component({
  selector: 'pomodoro-app',
  directives: [ROUTER_DIRECTIVES],
  providers: [SHARED_PROVIDERS, HTTP_PROVIDERS, ROUTER_PROVIDERS],
  template: `
...
```

Настройка службы маршрутизации

После объявления поставщиков и директив необходимо превратить главный компонент в компонент маршрутизатора. В принципе, любой компонент можно превратить в компонент маршрутизации, удовлетворив следующие требования:

- помимо декоратора `@Component`, класс компонента должен быть отмечен декоратором `@RouteConfig`;
- декоратор `@RouteConfig` настраивается с помощью массива объектов `RouteDefinition`, которые, в сущности, являются литералами объектов путей, идентифицируемых с помощью имени и указателя на тип компонента;
- затем компонент, отмеченный декоратором `@RouteConfig`, следует включить в директиву `RouterOutlet` в его шаблоне. Этот элемент станет заполнителем, куда будут загружены и где будут выводиться компоненты после загрузки маршрута, указывающего на них. При этом будут удалены все предыдущие компоненты, если таковые имеются.

В этом смысле можно сказать, что маршрутизатор отслеживает изменение адресной строки в браузере и выполняет поиск объекта `RouteDefinition`, чье свойство `path` соответствует текущему URL-адресу. Затем создает экземпляр компонента, указанного в определении маршрута внутри заполнителя, предоставленного директивой `RouterOutlet` в шаблоне, относящемся к компоненту, декорированному с помощью `@RouteConfig`.

Рассмотрим все это на реальном примере. Как уже упоминалось во введении к главе, наше приложение нуждается в улучшении архитектуры навигации, чтобы сделать его более удобным и интуитивно понятным. После распределения логики по компонентам в *главе 5 «Создание приложений с помощью компонентов Angular 2»* определим маршруты, реализующие следующую логику.

- Пользователь входит в приложение и просматривает текущий список заданий, ожидающих выполнения. Он должен иметь возможность планировать будущие задания, чтобы оценить требуемое время их выполнения в течение следующего Pomodoro-сеанса.
- При желании пользователь может перейти на страницу с формой создания задания (здесь будет создана лишь сама форма, а функции редактирования будут реализованы в следующей главе).
- Пользователь всегда может выбрать любое задание и запустить Pomodoro-сеанс для его выполнения.
- Пользователь может перемещаться назад и вперед по ранее посещавшимся страницам.

Создание нового компонента для демонстрационных целей

На настоящий момент созданы два самостоятельных компонента, которые можно использовать для организации межстраничной навигации. Но для удобного взаимодействия с пользователем может потребоваться еще и третий. Для расширения возможностей навигации мы добавим компонент формы, который более тщательно будет проработан в *главе 8 «Формы и аутентификация в Angular 2»*.

Создадим в папке `tasks` компонент, определяющий форму, которая будет использоваться в следующей главе для публикации новых заданий. Для этого создайте следующие файлы:

APP/TASKS/TASK-EDITOR.COMPONENT.TS

```
import { Component } from '@angular/core';
import { ROUTER_DIRECTIVES } from '@angular/router-deprecated';

@Component({
  selector: 'pomodoro-tasks-editor',
  directives: [ROUTER_DIRECTIVES],
  templateUrl: 'app/tasks/task-editor.component.html'
})
export default class TaskEditorComponent {
  constructor() {}
}
```

APP/TASKS/TASK-EDITOR.COMPONENT.HTML

```
<form class="container">
  <h3>Task Editor:</h3>
  <div class="form-group">
    <input type="text"
      class="form-control"
      placeholder="Task name"
      required>
</div>

<div class="form-group">
  <input type="Date"
    class="form-control"
    required>
</div>

<div class="form-group">
```

```
    <input type="number"
      class="form-control"
      placeholder="Pomodoros required"
      min="1"
      max="4"
      required>
  </div>

  <div class="form-group">
      <input type="checkbox" name="queued">
      <label for="queued"> this task by default?</label>
    </div>

    <p>
      <input type="submit" class="btn btn-success" value="Save">
      <a href="/" class="btn btn-danger">Cancel</a>
    </p>
</form>
```

Это базовое определение компонента, но в него добавлен псевдоним ROUTER_DIRECTIVES из библиотеки маршрутизатора. В дальнейшем это обеспечит подключение директивы маршрутизации в HTML-шаблоне. Она будет использоваться для добавления в шаблон ссылок перехода к другим компонентам, как будет показано в ближайшее время. И последнее, но важное замечание: этот новый компонент нужно экспортировать в фасаде:

APP/TASKS/TASKS.TS

```
import TasksComponent from './tasks.component';
import TaskEditorComponent from './task-editor.component';
import TaskTooltipDirective from './task-tooltip.directive';

const TASKS_DIRECTIVES: any[] = [
  TasksComponent,
  TaskEditorComponent,
  TaskTooltipDirective
];

export {
  TASKS_DIRECTIVES,
  TasksComponent,
  TaskEditorComponent,
  TaskTooltipDirective
};
```

Настройка декоратора RouteConfig с помощью экземпляров RouteDefinition

Для достижения поставленных целей необходимо создать маршрутизатор верхнего уровня, который будет отвечать за определение маршрутов. Логический путь начинается с корневого компонента. Откройте файл модуля и импортируйте следующие имена, непосредственно за импортом ROUTER_PROVIDERS:

APP/APP.COMPONENT.TS

```
...
import {
  ROUTER_PROVIDERS,
  RouteConfig,
  ROUTER_DIRECTIVES
} from '@angular/router-deprecated';
import { TimerComponent } from './timer/timer';
import {
TasksComponent,
TaskEditorComponent } from './tasks/tasks';
...
```

RouteConfig представляет тип декоратора, который превратит этот компонент в компонент маршрутизатора. Псевдоним ROUTER_DIRECTIVES включает директивы представлений, которые вскоре потребуются нам для оформления ссылок на маршруты. Импортируем также все три компонента, с которыми будет вестись работа. Как будет показано ниже, для каждого маршрута необходимо также объявить тип компонента.

Теперь заменим директивы в модуле AppComponent символом ROUTE_DIRECTIVES, так как отпала необходимость объявлять в фасаде компоненты, размещенные в шаблоне. Маршрутизатор сделает это сам:

APP/APP.COMPONENT.TS

```
@Component({
  selector: 'pomodoro-app',
  directives: [ROUTER_DIRECTIVES],
  providers: [SHARED_PROVIDERS, HTTP_PROVIDERS, ROUTER_PROVIDERS],
  template: `
  ...
})
```

А теперь расширим определение класса компонента декоратором RouteConfig, добавив его сразу после декоратора @Component, непосредственно перед оператором class:

APP/APP.COMPONENT.TS

```
@RouteConfig([
  { path: '',
    name: 'TasksComponent',
    component: TasksComponent
  }, {
    path: 'tasks/editor',
    name: 'TaskEditorComponent',
    component: TaskEditorComponent
  }, {
    path: 'timer',
    name: 'TimerComponent',
    component: TimerComponent
  }
])
export default class AppComponent {}
```

Как уже отмечалось в начале главы, декоратор RouteConfig следует заполнить массивом объектов RouteDefinition, каждый из которых определяет путь, куда может перейти пользователь. Это позволит пользователю обратиться к компоненту, тип которого определен в свойстве component.



Примечание: в новом маршрутизаторе декоратор @RouteConfig заменен декоратором @Routes. Свойство name удалено из схемы определения маршрутов, и сопоставление маршрутов выполняется на основании путей.

В примере корневой компонент будет обслуживать три различных маршрута к элементам TasksComponent, TimerComponent и TaskEditorComponent.



Здесь вступает в действие другое общее соглашение для предыдущей версии маршрутизатора Angular 2: имена маршрутов должны совпадать с именами компонентов, к которым они относятся. Как вскоре будет показано, имена маршрутов используются при заполнении ссылок на ресурсы, поэтому совпадение их с именами компонентов весьма полезно с точки зрения интуитивного определения целей ссылок в шаблоне. Следовать этому соглашению в новом маршрутизаторе необязательно, поскольку там при сопоставлении маршрутов используются только пути.

На данный момент остались нерешенными два вопроса: где будут отображаться компоненты и как будут активироваться маршруты? Для получения ответов на эти вопросы подробно рассмотрим директивы маршрутизатора.

Директивы маршрутизатора RouterOutlet и RouterLink

Псевдоним `ROUTER_DIRECTIVES` предоставляет доступ лишь к двум необходимым директивам.

Первая из них – `RouterOutlet` – служит заполнителем, где будут отображаться различные компоненты, в зависимости от пути перехода. Директива `RouterLink` используется в качестве директивы-атрибута. Она придает HTML-элементам поведение ссылок или кнопок для перехода по различным маршрутам, определяя уникальные имена этих маршрутов, как будет показано ниже.

В предыдущих главах был подготовлен шаблон корневого компонента, отображающий привлекательный заголовок панели навигации `nav`, ниже которого располагаются элементы с Pomodoro-таймером и списком заданий. Теперь выделим HTML-шаблон из определения декоратора компонента и сохраним его в отдельном файле, чтобы упростить к нему доступ, когда потребуется его отредактировать. Кроме того, внесем в шаблон компонента изменения, связанные с маршрутизацией, добавив ссылки на различные представления или состояния, определяющие возможности приложения:

APP/APP.COMPONENT.TS

```
 ...
 @Component({
selector: 'pomodoro-app',
directives: [ROUTER_DIRECTIVES],
providers: [SHARED_PROVIDERS, HTTP_PROVIDERS, ROUTER_PROVIDERS],
templateUrl: 'app/app.component.html'
})
...
```

APP/APP.COMPONENT.HTML

```
<nav class="navbar navbar-default navbar-static-top">
  <div class="container">
    <div class="navbar-header">
      <strong class="navbar-brand">My Pomodoro App</strong>
    </div>
```

```
<ul class="nav navbar-nav navbar-right">
  <li><a [routerLink]="['TasksComponent']">Tasks</a></li>
  <li><a [routerLink]="['TimerComponent']">Timer</a></li>
  <li>
    <a [routerLink]="['TaskEditorComponent']">
      Publish Task
    </a>
  </li>
    </ul>
  </div>
</nav>

<router-outlet></router-outlet>
```

Как видите, там, где раньше в шаблоне располагались компоненты, теперь находятся директива `<router-outlet>` и три новые ссылки, составляющие привлекательную навигационную панель `nav`. Перезагрузите страницу и посмотрите, как выводится список заданий, а затем щелкните на ссылке **Timer**. Здорово! На страницу загрузился элемент `TimerComponent`. Щелкните на другой ссылке или кнопке возврата в браузере, чтобы убедиться в возможности переходов между компонентами.

Рассмотрим эти ссылки поближе, выбрав для примера ссылку на таймер:

```
<a [routerLink]="['TimerComponent']">Timer</a>
```

Морфология директивы `routerLink` не нуждается в особых пояснениях. Справа от символа равенства указан массив, в котором первым следует имя маршрута, а за ним – имена дополнительных маршрутов для навигации в пределах данного маршрута. Обычно используется лишь одно уникальное строковое значение. Однако массив может содержать несколько значений, если компонент, на который указывает ссылка, имеет собственный маршрутизатор с именами маршрутов. В этом случае в массив строк добавляются имена дополнительных маршрутов. Именно поэтому считается удобным и рекомендуется называть маршруты именами компонентов, на которые они указывают.



Новый маршрутизатор не использует имена маршрутов, отдавая предпочтение URL-путям. То есть при использовании нового маршрутизатора в директиве `[routerLink]` следует указывать полный путь.

Кроме того, определение `RouterLink` позволяет декларативно добавлять параметры, поэтому маршруты можно динамически менять пря-

мо во время выполнения. Все эти возможности будут использоваться в следующих разделах, но уже теперь нужно получить ответ на один важный вопрос. Как осуществить переход к компоненту императивно, не по щелчку на ссылке, а в результате побочного действия, выполняемого в рамках класса контроллера компонента?

Принудительный переход по маршрутам

Возможно, пользователю понадобится перейти к таймеру, выбрав задание, над которым он хотел бы поработать. Ранее уже была реализована функция, отображающая метку очереди для каждого задания, выбираемого для выполнения. Используем тот же подход для имитации щелчка на кнопке, перенаправляющего пользователя на компонент таймера.

Во-первых, добавим тип `Router` как зависимость в модуль `TasksComponent`, чтобы получить программный доступ к его методам. Поскольку во время начальной загрузки приложения уже был объявлен символ `ROUTER_PROVIDERS`, фреймворк Angular 2 сам позаботится о внедрении типа `Router`, если надлежащим образом объявить его в нашем компоненте. Для этого добавьте следующий оператор импорта в начало компонента вслед за другими операторами импорта:

APP/TASKS/TASKS.COMPONENT.TS

```
import { Router } from '@angular/router-deprecated';
```

А теперь внедрим тип `Router` через конструктор. Отметим его в конструкторе как `private`, чтобы сделать доступным только внутри компонента:

APP/TASKS/TASKS.COMPONENT.TS

```
...
constructor(
  private taskService: TaskService,
  private settingsService: SettingsService,
  private router: Router) {
    this.tasks = this.taskService.taskStore;
    this.today = new Date();
    this.queueHeaderMapping = settingsService.pluralsMap.tasks;
    this.timerMinutes = settingsService.timerMinutes;
  }
...
```

А теперь ниже метода `updateQueuedPomodoros()` добавим новый метод, программно перенаправляющий пользователя по маршруту для выполнения задания:

```
workOn(): void {
  this.router.navigate(['TimerComponent']);
}
```

Но как его вызвать? Добавим в последнюю ячейку каждой строки таблицы новую кнопку, рядом с кнопкой переключения заданий, обработчик щелчка которой указывает на метод `workOn()`:

APP/TASKS/TASKS.COMPONENT.HTML

```
<td>
  <button type="button" class="btn btn-default btn-xs"
    [ngSwitch]="task.queued"
    (click)="toggleTask(task)">
    <template [ngSwitchWhen]="false">
      <i class="glyphicon glyphicon-plus-sign"></i>
      Add
    </template>
    <template [ngSwitchWhen]="true">
      <i class="glyphicon glyphicon-minus-sign"></i>
      Remove
    </template>
    <template ngSwitchDefault>
      <i class="glyphicon glyphicon-plus-sign"></i>
      Add

    </template>
  </button>
  <button type="button"
    class="btn btn-default btn-xs"
    ngIf="task.queued"
    (click)="workOn()">
    <i class="glyphicon glyphicon-expand"></i> Start
  </button>
</td>
```

Директива `NgIf` отобразит кнопку только при возникновении такой необходимости. Значок добавлен лишь для украшения, можете удалить его или заменить на любой другой по своему вкусу.

А теперь перезагрузите таблицу, выберите любое задание, которое предстоит выполнить, и щелкните на появившейся кнопке. Вуаля! Вы будете перенаправлены на таймер, чтобы иметь возможность начать работу над этим заданием.



Метод `navigate()` в новом маршрутизаторе требует передачи строки, содержащей полный путь.

CSS-ловушки для активных маршрутов

Мы видели, как превратить любую ссылку или элемент DOM в гиперссылку, указывающую на именованный маршрут, который создает экземпляр компонента. Однако неплохо было бы реализовать и некое визуальное выделение активной ссылки. Директива `routerLink` дает такую возможность. Всякий раз, когда маршрут, указанный в директиве `routerLink`, становится активным (в результате действий пользователя или выполнения программных операций), в элемент добавляется CSS-класс `router-link-active`. Поэтому в таблицах стилей компонентов можно определить специальные стили для выделения активных ссылок.

Для этого потребуется лишь небольшая настройка корневого компонента. Откройте файл класса контроллера корневого компонента и добавьте следующие стили в конфигурацию декоратора компонента:

APP/APP.COMPONENT.TS

```
@Component({
  selector: 'pomodoro-app',
  directives: [ROUTER_DIRECTIVES],
  providers: [SHARED_PROVIDERS, HTTP_PROVIDERS, ROUTER_PROVIDERS],
  styles: [`
      .router-link-active {
          font-weight: bold;
          border-bottom: 2px #d9534f solid;
      }
  `],
  templateUrl: 'app/app.component.html'
})
```

А теперь перезагрузите браузер и щелкните на любой ссылке или перейдите к таймеру из таблицы заданий, следя за внешним состоянием верхней навигационной панели. К активной ссылке будет применен определенный выше стиль. Заглянув в код разметки, можно убедиться в применении класса `router-link-active` к активной ссылке.

Параметры обработки маршрута

До сих пор выполнялась настройка только основных путей маршрутов, а можно ли динамически создавать пути с поддержкой параметров или значений, вычисленных во время выполнения? Создание

URL-адреса (и переход по нему) для загрузки определенных данных из хранилища – типичная задача, которую приходится решать ежедневно. К примеру, может потребоваться реализовать отношение «один ко многим», чтобы URL-адрес, сгенерированный на главной странице, содержал идентификаторы элементов для загрузки на странице с подробными сведениями.

Здесь имеются две основные трудности: создание URL-адресов с динамическими параметрами непосредственно во время выполнения и анализ значений этих параметров. Но при использовании маршрутизатора Angular это не является проблемой, как будет показано на реальном примере.

Передача динамических параметров в маршрутах

Мы добавили в список заданий кнопку для перехода на страницу компонента таймера. Но при загрузке компонент таймера не получает никакого контекста, позволяющего определить, какое из заданий предстоит выполнить. Доработаем компонент и добавим в него отображение задания, выбранного до перехода на эту страницу.

Во-первых, вернемся в шаблон компонента списка заданий и исправим объявление кнопки, вызывающей переход к компоненту таймера, добавив в него индекс элемента задания, соответствующего текущей итерации цикла:

APP/TASKS/TASKS.COMPONENT.HTML

```
...
  <button type="button"
    class="btn btn-default btn-xs"
    *ngIf="task.queued"
    (click)="workOn(i)">
    <i class="glyphicon glyphicon-expand"></i> Start
  </button>
...
```

Напомню, что индекс создается в каждой итерации директивы `NgFor`, отображающей строки таблицы. Теперь, когда индекс передается в вызов метода, осталось лишь изменить определение самого метода:

```
workOn(index: number): void {
  this.router.navigate(['Timer', { id: index }]);
}
```

В директиве routerLink параметры можно определить в виде объекта хэша, следующего в массиве за строкой с именем пути (или несколькими строками, как мы увидим ниже, в обсуждении дочерних маршрутизаторов). Точно так же добавляются параметры в динамически сгенерированные ссылки. Однако если теперь щелкнуть на любой кнопке, динамическое значение идентификатора добавится как параметр строки запроса. В некоторых случаях этого вполне достаточно, но существует более элегантное решение. Поэтому изменим определение маршрута, чтобы включить параметр в путь. Вернемся в корневой компонент и исправим маршрут в декораторе RouteConfig, как показано ниже:

APP/APP.COMPONENT.TS

```
...
}, {
  path: 'timer/:id',
  name: 'TimerComponent',
  component: TimerComponent
}
...
```

Перезагрузим приложение, выберем последнее задание в таблице и щелкнем на кнопке **Start**. Как видите, браузер загрузил компонент Timer по URL-адресу /timer/3.

Любой путь может содержать столько параметров с двоеточием, сколько потребуется. Параметры будут замещаться актуальными значениями при выполнении директивы routerLink или метода navigate класса Router, передачей хэша с парами ключ/значение, соответствующими параметрам. Итак, в общем случае пути маршрутов можно определить следующим образом:

```
{
  path: '/products/:category/:id',
  name: 'ProductsByCategoryComponent',
  component: ProductsByCategoryComponent
}
```

Тогда можно будет выполнить переход по любому маршруту, подобно тому, как было продемонстрировано выше, следующим образом:

```
<a [routerLink]="['ProductsByCategoryComponent', {
  category: 'toys',
  id: 452
}]">See Toy</a>
```

То же относится к переходам по маршрутам программным способом:

```
router.navigate(['ProductsByCategoryComponent', {
  category: 'toys',
  id: 452
}]);
```

Разбор параметров маршрута с помощью службы RouteParams

Отлично! Теперь имеется возможность передать индекс элемента `task`, для которого следует загрузить таймер, но как выделить этот параметр из URL-адреса? Маршрутизатор Angular предоставляет удобный внедряемый тип `RouteParams` (он уже включен в псевдоним `ROUTER_PROVIDERS`), который можно использовать в компонентах, обрабатываемых маршрутизатором, для извлечения параметров, указанных в определении пути маршрута.

Откроем компонент `timer` и импортируем этот тип, как показано ниже, а также внедрим поставщика `TaskService`, чтобы можно было получать информацию из элемента списка заданий:

APP/TIMER/TIMER-WIDGET.COMPONENT.TS

```
import { Component, OnInit } from '@angular/core';
import { SettingsService, TaskService } from '../shared/shared';
import { RouteParams } from '@angular/router-deprecated';
...
```

Мы должны изменить определение компонента и определить тип `TaskService` как аннотируемую зависимость, чтобы инструмент внедрения зависимостей смог отыскать соответствующего поставщика.

Новый маршрутизатор не использует класс `RouteParams`, вместо него следует задействовать новый класс `RouteSegments`, имеющий больше полезных методов. Подробное описание программного интерфейса этого класса можно найти в официальной документации.

Дополнительно вставим в шаблон компонента заголовок для отображения названия задания:

APP/TIMER/TIMER-WIDGET.COMPONENT.TS

```
...
@Component({
  selector: 'pomodoro-timer-widget',
  template: `
```

```
<div class="text-center">
  <img src="/app/shared/assets/img/pomodoro.png">
  <h3><small>{{ taskName }}</small></h3>
  <h1> {{ minutes }}:{{ seconds | number: '2.0' }} </h1>
  <p>
    <button (click)="togglePause()" class="btn btn-danger">
      {{ buttonLabelKey | i18nSelect: buttonLabelsMap }}
    </button>
  </p>
</div>`
})
...
```

В заголовке используется интерполируемое выражение с переменной `taskName`, хранящей название задания. Опираясь на уже сделанное, внесем изменения в конструктор для ввода в игру типа `RouteParams` и класса `TaskService` в качестве закрытых свойств, внедряемых в конструкторе:

APP/TIMER/TIMER-WIDGET.COMPONENT.TS

```
...
constructor(
  private settingsService: SettingsService,
  private routeParams: RouteParams,
  private taskService: TaskService) {
    this.buttonLabelsMap = settingsService.labelsMap.timer;
}
...
```

После того как эти типы стали доступны в классе, можно добавить в обработчик `ngOnInit` извлечение подробных сведений об элементе массива заданий с индексом, указанным в параметре. Задержка в `OnInit` выполняется не случайно – поскольку возникнут проблемы при попытке обратиться к свойствам `routeParams` до ее завершения:

APP/TIMER/TIMER-WIDGET.COMPONENT.TS

```
ngOnInit(): void {
  this.resetPomodoro();
  setInterval(() => this.tick(), 1000);

  let taskIndex = parseInt(this.routeParams.get('id'));
  if (!isNaN(taskIndex)) {
    this.taskName = this.taskService.taskStore[taskIndex].name;
  }
}
```

Как извлечь значение из параметра `id`? Объект `RouteParams` экспортирует метод `get(param: string)`, принимающий имя параметра. В примере выше мы извлекаем значение параметра `id` с помощью вызова `routeParams.get('id')`. Обычно именно так извлекаются параметры из маршрутов. Сначала нужно создать экземпляр класса `RouteParams` с помощью инструмента внедрения зависимостей компонента, а затем извлечь нужное значение, вызвав метод `get` экземпляра и передав ему строку с именем соответствующего параметра маршрута.



Важно отметить, что данные, извлеченные ранее, были сохранены в свойстве `taskStore` поставщика `TaskService`. Поскольку это синглтон, доступность которого во всем приложении обеспечивает механизм внедрения зависимостей фреймворка Angular, и он заранее был заполнен в компоненте `TaskComponent`, вся необходимая информация уже имеется. Реализовать загрузку данных внутри таймера было бы намного сложнее. Для этого потребовалось бы подписаться на службу, чтобы принудительно загрузить данные через Http-клиента, а для интерполяции `taskName` в шаблоне потребовалось бы использовать фильтр `async`, как было описано в *главе 6* «*Асинхронная обработка данных в Angular 2*». Для простоты этот случай здесь не рассматривается, но я рекомендую самостоятельно настроить компонент для поддержки и этого случая.

Определение дочерних маршрутизаторов

С точки зрения масштабируемости приложений, объединение всех определений маршрутов в одном месте (например, в корневом компоненте) не выглядит удачной идеей. Чем больше маршрутов определено, тем труднее поддерживать приложения, не говоря уже об образовании жестких связей между компонентами (которые предназначены для многократного использования и не должны зависеть от конкретного приложения, насколько это возможно) и самим приложением.

Именно поэтому предпочтительнее выделять и изолировать определения маршрутов к отдельным функциям в конфигурациях маршрутизаторов конкретных компонентов, а на долю корневого компонента оставлять только обертывание контекстов функций. Команда Angular руководствовалась именно этой идеей при разработке библиотеки `Router` и реализовала поддержку дополнения базового маршрута дочерними, при сохранении родительского маршрута полностью независимым от влияния маршрутов на верхних уровнях.

Рассмотрим это на реальном примере. Последнее, что мы сделали, – добавили в таймер отображение названия задания, выбранного

в списке. А что, если потребуется использовать таймер сам по себе, без связи с каким-либо заданием? Такая возможность позволит использовать таймер для экспромт-заданий, без необходимости создавать их заранее.

То есть необходимо обеспечить два режима использования таймера:

- `timer`: без конкретного задания;
- `timer/task/{id}`: с заданием, где `{id}` – индекс задания в списке.

Начнем с внесения изменений в корневой компонент, ставший теперь компонентом маршрутизатора, чтобы преобразовать путь `/timer/:id` в путь, указывающий на дочерний компонент маршрутизатора. Откройте компонент и замените определение маршрута, указывающего на компонент таймера, следующим определением:

APP/APP.COMPONENT.TS

```
{
  path: 'timer/...',
  name: 'TimerComponent',
  component: TimerComponent
}
```

Вот и все. Многоточие в конце маршрута указывает маршрутизатору `Router` в Angular, что он должен учитывать определения маршрутов во вложенном компоненте. Проблема здесь заключается в том, что компонент не может служить маршрутизатором самого себя, так как он не может создать свой собственный экземпляр в своей директиве `RouterOutlet`. Вот почему для этого примера необходим компонент маршрутизатора, представляющий компонент таймера. Итак, для простоты создадим компонент маршрутизации для таймера в той же папке, где он расположен. Компонент маршрутизации по определению является компонентом, не делающим ничего другого, кроме выполнения обязанностей диспетчера маршрутов. Его реализация, как правило, ограничивается декоратором `RouteConfig`, содержащим маршруты к функциям, предлагаемым компонентом, и включением директивы `RouterOutlet` в его шаблон.

Компоненты маршрутизации являются весьма удобным способом отделения маршрутизации от реализаций компонентов, обеспечивающим полноценное многократное их использование.

Откройте папку `timer` и создайте файл для компонента маршрутизации таймера:

APP/TIMER/TIMER.COMPONENT.TS

```
import { Component } from '@angular/core';
import { RouteConfig, ROUTER_DIRECTIVES } from '@angular/router-deprecated';
import TimerWidgetComponent from './timer-widget.component';

@Component({
  selector: 'pomodoro-timer',
  directives: [ROUTER_DIRECTIVES],
  template: '<router-outlet></router-outlet>'
})
@RouteConfig([
  { path: '/', name: 'GenericTimer',
    component: TimerWidgetComponent,
    useAsDefault: true
  }, {
    path: '/task/:id',
    name: 'TaskTimer',
    component: TimerWidgetComponent
  }
])
export default class TimerComponent {}
```

Как видите, вновь созданный класс компонента не содержит ничего, кроме диспетчера маршрутов для `TimerWidgetComponent`. Тип `RouteConfig` позволяет создать нужный декоратор, содержащий определения маршрутов, а псевдоним `ROUTER_DIRECTIVES` обеспечивает внедрение директивы `RouterOutlet` в шаблон компонента.

Обратите внимание на новое логическое свойство `useAsDefault` в определении первого маршрута. Оно сообщает маршрутизатору, что если не найдено подходящего пути, класс `Router` должен перейти по этому маршруту.



На момент написания этих строк новый маршрутизатор все еще не имел свойства `useAsDefault`, но его поддержка планируется в окончательной версии. Более подробную информацию можно найти в официальной документации.

Важно отметить, что любой компонент, служащий маршрутизатором, может иметь собственную реализацию в директиве `RouterOutlet`. Подобным способом, как это реализовано в `AppComponent`, можно добавлять в компонент дополнительные функциональные возможности, помимо маршрутизации.

Итак, мы получили компонент, способный отправлять пользователей к компоненту таймера двумя способами, но как на него ссылаться?

Ссылки на дочерние маршруты

Нет никакой разницы между ссылками на дочерний маршрутизатор и ссылками на любой маршрут, управляемый маршрутизатором верхнего уровня, за исключением того, что массив имен маршрутов заполняется именами дочерних маршрутов. Откройте компонент `PomodoroTaskList` и измените метод `workOn()`, как показано ниже:

```
workOn(index: number): void {
  this.router.navigate(['TimerComponent', 'TaskTimer', {
    id: index
  }]);
}
```

Здесь для фреймворка Angular задается ссылка на маршрут `TimerComponent` (поэтому так важно давать маршрутам имена целевых компонентов). Так как это родительский маршрут (помните про многоточие), заданный в корневом маршрутизаторе, для перехода по маршрутам, объявленным в дочернем маршрутизаторе, необходимо указать имя дочернего маршрута – в данном случае в `TaskTimer`. Очевидно, что к маршруту должна быть присоединена информация об идентификаторе, необходимая для загрузки нужного задания. Щелкните на любом задании, и вы увидите, как загрузится таймер и отобразится имя выбранного задания.

Такой подход к реализации позволяет выводить компонент как с информацией о задании, так и без нее. То есть на страницу таймера можно перейти либо по ссылке в верхней навигационной панели, либо щелчком на кнопке **Start** в таблице заданий.

Помните, как при настройке главного маршрута мы присвоили свойству `useAsDefault` значение `true`? Это означает, что в отсутствие идентификатора задания в маршруте к таймеру будет выбрано определение этого маршрута.

События жизненного цикла маршрутизатора

По аналогии с компонентами, преодолевающими разные этапы в течение своей жизни, жизненный цикл маршрутизаторов также делится на несколько этапов. На каждом из них генерируются определенные события, которые, так же как в случае с компонентами, можно обрабатывать с помощью реализаций конкретных интерфейсов в компонентах. Единственным исключением из этого правила является самое

раннее событие CanActivate, обработчик которого имеет форму декоратора, поскольку должен вызываться еще до создания компонента.

Событие CanActivate

Обработчик события CanActivate, определяемый в виде декоратора компонента, вызывается маршрутизатором Router непосредственно перед созданием экземпляра компонента. Это должна быть функция, возвращающая логическое значение (или объект Promise с логическим значением), указывающее на необходимость активации компонента:

```
@CanActivate((next, prev) => boolean | Promise<boolean>)
```

То есть декоратор @CanActivate – это функция, принимающая другую функцию в аргументе, которая, в свою очередь, принимает в виде параметров два объекта ComponentInstruction: первый представляет маршрут для перехода, а второй – маршрут, откуда выполняется переход. Эти объекты экспортируют полезные свойства, описывающие маршрут, откуда выполняется переход, и компонент, экземпляр которого следует создать: путь, параметры, тип компонента и т. д.



Обработчик этого события – отличное место для реализации таких действий, как проверка допустимости сеанса, что позволяет защитить отдельные области приложения. К сожалению, CanActivate не имеет встроенной поддержки внедрения зависимостей, что усложняет внедрение расширенной логики перед активацией маршрута. В следующих главах будут рассмотрены способы обхода этого ограничения для таких действий, как проверка подлинности пользователя.

Следующий пример демонстрирует защиту формы паролем, которая препятствует ее созданию, если пользователь введет неверный пароль. Во-первых, откройте файл модуля TaskEditorComponent и импортируйте все, что потребуется для первого эксперимента, а также все псевдонимы, необходимые для реализации обработчиков событий жизненного цикла маршрутизаторов. Затем примените декоратор CanActivate к классу компонента:

APP/TASKS/TASK-EDITOR.COMPONENT.TS

```
import { Component } from '@angular/core';
import {
  ROUTER_DIRECTIVES,
  CanActivate,
  ComponentInstruction,
  OnActivate,
```

```
  CanDeactivate,
  OnDeactivate } from '@angular/router-deprecated';

@Component({
  selector: 'pomodoro-tasks-editor',
  directives: [ROUTER_DIRECTIVES],
  templateUrl: 'app/tasks/task-editor.component.html'
})
@CanActivate((
  next: ComponentInstruction,
  prev: ComponentInstruction): boolean => {
    let passPhrase = prompt('Say the magic words');
    return (passPhrase === 'open sesame');
  }
)
export default class TaskEditorComponent {
...
```

Как видите, декоратор `@CanActivate` содержит стрелочную функцию с двумя аргументами типа `ComponentInstruction` (которые не нужны в данном примере и приведены исключительно в учебных целях). Стрелочная функция возвращает логическое значение – результат сравнения ввода пользователя с правильным паролем, с учетом регистра букв. Просмотрите в консоли значения параметров `next` и `prev`, чтобы лучше познакомиться с информацией, предоставляемой этими двумя полезными объектами.

Кстати, обратили внимание на присутствие псевдонима `ROUTER_DIRECTIVES` в свойстве `directives`? В данном обзоре событий жизненного цикла директивы маршрутизации нам не понадобятся, но они понадобятся позднее, в дальнейших экспериментах с разными событиями жизненного цикла. Теперь добавим для удобства кнопку возврата, используя для этого кнопку **Cancel**, уже присутствующую в шаблоне компонента:

APP/TASKS/TASK-EDITOR.COMPONENT.HTML

```
<form class="container">
...
  <p>
    <input type="submit" class="btn btn-success" value="Save">
    <a [routerLink]="['TasksComponent']" class="btn btn-danger">
      Cancel
    </a>
  </p>
</form>
```

Событие OnActivate

Событие `OnActivate` позволяет выполнить дополнительные действия сразу после успешного перехода по маршруту к компоненту. Его легко обработать, реализовав простой интерфейс. В обработчике допускается выполнять даже асинхронные операции, достаточно лишь вернуть объект `Promise` из функции интерфейса. При таком подходе маршрут изменится только после разрешения объекта `Promise`.

Рассмотрим это на реальном примере, суть которого заключается в изменении заголовка страницы. Для этого продолжим работу с модулем `TaskEditorComponent`, куда добавим поддержку интерфейса события `OnActivate` и класса `Title` (https://angular.io/docs/ts/latest/api/platform/browser/Title-class.html), имеющего методы для записи и чтения заголовка страницы при выполнении приложения в веб-браузере. Импортируем символ `Title` и объявим его в компоненте как поставщика, сделав доступным для инструмента внедрения зависимостей (этот объект можно внедрить в корневой компонент, если планируется использовать его в других компонентах):

APP/TASKS/TASK-EDITOR.COMPONENT.TS

```
import { Component } from '@angular/core';
import {
  ROUTER_DIRECTIVES,
  CanActivate,
  ComponentInstruction,
  OnActivate,
  CanDeactivate,
  OnDeactivate } from '@angular/router-deprecated';
import { Title } from '@angular/platform-browser';

@Component({
  selector: 'pomodoro-tasks-editor',
  directives: [ROUTER_DIRECTIVES],
  providers: [Title],
  templateUrl: 'app/tasks/task-editor.component.html'
})
```

А теперь реализуем интерфейс, определив обязательный метод `routerOnActivate`. Как правило, всем обработчикам событий жизненного цикла маршрутизатора присваиваются имена, состоящие из имени события и префикса `router`, записанного в нижнем регистре:

APP/TASKS/TASK-EDITOR.COMPONENT.TS

```
export default class TaskEditorComponent implements OnActivate {

  constructor(private title: Title) {}
  routerOnActivate(
    next: ComponentInstruction,
    prev: ComponentInstruction): void {
      this.title.setTitle('Welcome to the Task Form!');
    }
}
```

Обратите внимание, как внедряется объект `Title` через конструктор класса и используется после активации маршрутизатора компонента в момент завершения перехода. Сохраните изменения и перезагрузите приложение. Как видите, заголовок окна браузера изменяется в случае успешного перехода к компоненту, после выполнения этапов `CanActivate` и `OnActivate`.

События CanDeactivate и OnDeactivate

По аналогии с реализацией операций при активации компонента, на который выполнен переход, имеется возможность реализовать некоторые действия, выполняемые, когда пользователь оставляет текущий компонент, переходя к другому. Как было показано выше, обработчик события `CanActivate` должен возвращать логическое значение или объект `Promise` с логическим значением, чтобы разрешить или запретить переход. Аналогично, когда обработчик `CanDeactivate` возвращает значение `true` или объект `Promise` со значением `true`, генерируется событие `OnDeactivate`, подобно тому, как генерируется событие `OnActivate` после завершения перехода.

Следующий пример реализует перехват этапов деактивации компонента маршрутизации, чтобы запросить у пользователя подтверждение на уход со страницы компонента и при получении согласия восстановить заголовок страницы. Для этого в компоненте нужно реализовать интерфейсы `CanDeactivate` и `OnDeactivate`:

```
export default class TaskEditorComponent implements OnActivate,
CanDeactivate, OnDeactivate {
  constructor(private title: Title) {}

  routerOnActivate(): void {
    this.title.setTitle('Welcome to the Task Form!');
  }
```

```
  routerCanDeactivate(): Promise<boolean> | boolean {
    return confirm('Are you sure you want to leave?');
  }

  routerOnDeactivate(): void {
    this.title.setTitle('My Angular 2 Pomodoro Timer');
  }
}
```

Обратите внимание на отсутствие в реализации аргументов (`next: ComponentInstruction, prev: ComponentInstruction`) – они не используются в данном примере, но с их помощью можно получить доступ к массе интересной информации, которая может пригодиться в других реализациях.

Подобно `CanActivate`, обработчик события `CanDeactivate` должен возвращать логическое значение или объект `Promise` с логическим значением, на основании которого принимается решение о возможности перехода.

События CanReuse и OnReuse

Наконец, один и тот же экземпляр компонента можно повторно использовать при переходе между компонентами одного типа. То есть можно сэкономить ресурсы, пропустив операции уничтожения и создания нового компонента.

Для этого требуется обеспечить надлежащее обновление информации в параметрах пользовательского интерфейса и, возможно, логики компонента в его новом воплощении.

За все эти действия отвечает обработчик события `CanReuse` и сообщает маршрутизатору `Router`, должен ли тот создать новый компонент или повторно использовать существующий при следующих переходах по тому же маршруту. Метод интерфейса `CanReuse` должен возвращать логическое значение или объект `Promise` с логическим значением (как это делают обработчики `CanActivate` и `CanDeactivate`), сообщающие маршрутизатору `Router` о возможности повторного использования этого компонента. Если реализация `CanReuse` возбудит ошибку или попытка будет отклонена в объекте `Promise`, переход будет отменен.

С другой стороны, если `CanReuse` вернет `true` или объект `Promise` будет разрешен как `true`, вместо обработчика `OnActivate` будет вызван обработчик `OnReuse`. Поэтому в своих реализациях используйте только одно из этих двух событий.

Рассмотрим все это на реальном примере. После выбора задания в списке и запуска таймера можно в любой момент перейти к универсальному таймеру, доступному из верхней навигационной панели, в результате чего запустится другой таймер, не связанный ни с каким заданием. В этом случае происходит переход от одного экземпляра компонента `TimerWidgetComponent` к другому, при этом маршрутизатор Angular уничтожает и вновь создает экземпляр того же компонента. Мы можем избавить маршрутизатор `Router` от этих действий, настроив компонент для повторного использования. Откройте модуль `TimerWidgetComponent` и импортируйте необходимые для этого интерфейсы вместе с символами, которые уже были ранее импортированы из библиотеки `Router`:

APP/TIMER/TIMER-WIDGET.COMPONENT.TS

```
import { Component, OnInit } from '@angular/core';
import { SettingsService, TaskService } from '../shared/shared';
import { RouteParams, CanReuse, OnReuse } from '@angular/router-deprecated';
```

А теперь реализуем интерфейсы `CanReuse` и `OnReuse`, включив следующие обязательные методы в тело класса:

```
routerCanReuse(): boolean {
  return true;
}

routerOnReuse(next: ComponentInstruction): void {
  // Еще не реализовано
}
```

Затем перейдите к таблице заданий, выберите любое задание и запустите таймер. Щелкните на ссылке **Timer** в верхней навигационной панели. Вы увидите, что URL-адрес в браузере изменился, но ничего не произошло, потому что повторно используется один и тот же компонент. Этот прием помогает экономить память, но требует обновить таймер после перехода. Внесем соответствующие изменения в метод `OnReuse`, сбросив значение `taskName` и сам таймер:

```
routerOnReuse(): void {
  this.taskName = null;
  this.isPaused = false;
  this.resetPomodoro();
}
```

А теперь повторим путешествие и посмотрим, что при этом произойдет. Вуаля! Новое поведение все того же компонента.

Дополнительные советы и хитрости

Мы обсудили все, что необходимо для создания сложных приложений с поддержкой маршрутизации, однако существует еще масса продвинутых методов, которые помогут поднять приложение на следующий уровень. Некоторые из них мы рассмотрим в следующих разделах.

Переадресация на другие маршруты

Помимо уже рассмотренных, имеется еще один тип определения маршрутов `RouteDefinition` – тип `Redirect`, предназначенный для переадресации на другой маршрут `Route`.

До сих пор за доступ к таблице со списком заданий отвечал только корневой маршрут. А можно ли получить доступ к этой таблице с использованием пути `/tasks` и при этом обеспечить правильную обработку всех ссылок, указывающих на корневой путь? Создадим маршрут переадресации. Внесем изменения в конфигурацию корневого маршрутизатора, добавив новый путь для существующего начального пути и путь переадресации на новый начальный URL-адрес:

APP/APP.COMPONENT.TS

```
...
@RouteConfig([{
  path: '',
  name: 'Home',
  redirectTo: ['TasksComponent']
}, {
  path: 'tasks',
  name: 'TasksComponent',
  component: TasksComponent,
  useAsDefault: true
}, {
  path: 'tasks/editor',
  name: 'TaskEditorComponent',
  component: TaskEditorComponent
}, {
  path: 'timer/...',
  name: 'TimerComponent',
  component: TimerComponent
}])
export default class AppComponent {}
```

В маршруте переадресации достаточно объявить строковое свойство пути `path` и свойство `redirectTo`, значением которого является массив именованных маршрутов для перенаправления всех запросов.



На момент написания книги поддержка свойства `redirectTo` в новом маршрутизаторе еще не была реализована. Более актуальную информацию по этому вопросу можно найти в официальной документации.

Точная настройка базового пути

Начиная работу над маршрутами в приложении, мы определили базовый адрес `href`, новый адрес и файл `index.html`, чтобы маршрутизатор мог находить любые загружаемые ресурсы, а не только компоненты. Корневой путь был задан явно, как /. Но что, если по некоторым причинам потребуется развернуть приложение по другому базовому URL-адресу, сохранив для представлений возможность находить все необходимые ресурсы, независимо от выбранного URL-адреса? Или, например, когда у нас нет доступа к тегу `HEAD` и мы не можем изменить тег `<base href="/">`, потому что разрабатываем распространяемый компонент и нам неизвестно, где этот компонент будет применяться? Независимо от причин, эту проблему легко решить, переопределив значение псевдонима `APP_BASE_HREF`, представляющего базовый путь `href`, который используется в типе `LocationStrategy`.

Попробуйте сами. Откройте файл `main.ts`, где выполняется начальная загрузка приложения, импортируйте необходимые псевдонимы и переопределите значение упомянутой выше переменной `href`, хранящей базовый путь приложения:

APP/MAIN.TS

```
import 'rxjs/add/operator/map';
import { bootstrap } from '@angular/platform-browser-dynamic';
import AppComponent from './app.component';
import { provide } from '@angular/core';
import { APP_BASE_HREF } from '@angular/common';

bootstrap(AppComponent, [provide(APP_BASE_HREF, {
  useValue: '/my-apps/pomodoro-app'
})]);
```

Перезагрузите приложение и посмотрите получившийся в результате URL-адрес в браузере.

Настройка генерируемых URL-адресов с учетом стратегии размещения

Как видите, после перехода по адресу, указанному в директиве `routerLink`, или в результате вызова метода `navigate` объекта `Router` адрес в адресной строке браузера соответствует привычным стандартизированным URL-адресам, но на самом деле это локальный URL-адрес, потому что фактически обращения к серверу не было. Отображение URL-адреса с естественной структурой обеспечивается методом `pushState` механизма истории в HTML5, вызываемого за кулисами и обеспечивающего прозрачность пополнения и изменения истории браузера.

Существуют два основных поставщика, наследующих тип `LocationStrategy` и предназначенных для представления и анализа состояния URL-адресов браузера:

- `PathLocationStrategy`: стратегия, используемая по умолчанию службой определения местоположения и отдающая предпочтение режиму `pushState` HTML5, который обеспечивает представление URL-адресов без хэш-фрагментов (`example.com/foo/bar/baz`);
- `HashLocationStrategy`: стратегия, использующая хэш-фрагменты для представления состояния в URL-адресе браузера (`example.com/#foo/bar/baz`).

Независимо от стратегии, выбранной по умолчанию службой `Location`, всегда можно вернуться к старой, основанной на хэшировании навигации, выбрав `HashLocationStrategy` в качестве `LocationStrategy`.

Для этого откройте файл `main.ts` и укажите глобальному инструменту внедрения зависимостей Angular, что с этого момента, когда потребуется внедрить экземпляр типа `LocationStrategy` для представления и анализа состояния (по умолчанию выбирается `PathLocationStrategy`), он должен использовать тип `HashLocationStrategy`.

Для этого достаточно переопределить поставщика зависимости по умолчанию:

APP/MAIN.TS

```
import 'rxjs/add/operator/map';
import { bootstrap } from '@angular/platform-browser-dynamic';
import AppComponent from './app.component';
import { provide } from '@angular/core';
import {
  LocationStrategy,
```

```
  HashLocationStrategy
} from '@angular/common';

bootstrap(AppComponent, [provide(LocationStrategy, {
  useClass: HashLocationStrategy
})]);
```

Сохраните изменения и перезагрузите приложение, запросив новый маршрут. Посмотрите на URL-адрес, получившийся в адресной строке браузера.



Обратите внимание, что псевдонимы, связанные с местоположением, в данном примере импортируются не из `@angular/router-deprecated`, а из `@angular/common`.

Асинхронная загрузка компонентов с помощью AsyncRoutes

Как было описано в этой главе, каждое определение маршрута необходимо настроить с помощью свойства `component`, которое будет информировать маршрутизатор о том, что загрузить, когда браузер получит определенный URL-адрес. Однако иногда компонент требуется получить во время выполнения или он является побочным продуктом асинхронной операции. В этих случаях следует применять различные способы выбора нужного компонента. Здесь поможет новый тип определения маршрутов `AsyncRoute`. Этот специальный вид маршрута имеет те же свойства, что и класс `RouteDefinition`, который уже использовался в этой главе. Он заменяет свойство `component` свойством `loader`, которое будет связано с объектом `Promise`, обеспечивающим асинхронную загрузку компонента по требованию.

Рассмотрим конкретный пример. Для простоты мы не будем импортировать компонент, загружаемый во время выполнения, а получим его с помощью асинхронной операции. Откройте модуль корневого компонента и измените маршрут, указывающий на `TimerComponent`, как показано ниже:

APP/APP.COMPONENT.TS

```
...

@RouteConfig([{
  path: '',
  name: 'Home',
  redirectTo: ['TasksComponent']
}, {
```

```
    path: 'tasks',
    name: 'TasksComponent',
    component: TasksComponent,
    useAsDefault: true
  }, {
    path: 'tasks/editor',
    name: 'TaskEditorComponent',
    component: TaskEditorComponent
  }, {
    path: '/timer/...',
    name: 'TimerComponent',
    loader: () => {
      return new Promise(resolve => {
        setTimeout(() => resolve(TimerComponent), 1000);
      });
    }
  }
])
export default class AppComponent {}
```

При следующей попытке выполнить переход по любому маршруту, принадлежащему ветви таймера (универсальный таймер, доступный из панели навигации, или таймер какого-либо конкретного задания), нужно будет ждать, пока `Promise` вернет нужный компонент. Цель этого примера – не в том, чтобы научить вас замедлять загрузку, а показать простой способ асинхронной загрузки компонентов.

Итоги

Эта глава была посвящена возможностям маршрутизатора Angular, и их описание должно помочь разобраться в тонкостях работы этой библиотеки. Одним из преимуществ модуля `Router` является огромное количество вариантов и сценариев, которые можно охватить с помощью простой, но мощной реализации.

В этой главе мы обсудили, как включить и настроить маршрутизацию в приложениях, как превратить любой компонент в компонент маршрутизации, добавив декоратор с настройками, как использовать маршруты в шаблоне и как распространить действие маршрутизаторов вниз по дереву компонентов. Мы также увидели, как определять обычные маршруты и маршруты некоторых продвинутых типов, такие как переадресация и асинхронные маршруты. Для нас не осталось секретов в жизненном цикле маршрутизации, и использование

его особенностей позволит реализовывать в приложении дополнительные функции без приложения особых усилий. Возможности бесконечны, и, что самое главное, маршрутизация улучшает восприятие приложения пользователями.

В следующей главе мы добавим компонент редактирования заданий для знакомства с механизмами Angular 2, лежащими в основе веб-форм, и способами извлечения данных, введенных пользователем с помощью элементов форм.

Глава 8

Формы и аутентификация в Angular 2

В предыдущей главе, где мы знакомились с механизмом маршрутизации, были упомянуты вопросы безопасности, когда речь зашла о предоставлении различных видов содержимого в приложении. Аутентификация пользователя является первым шагом в реализации определенных функций, таких как вывод форм в приложении. Однако для создания этих совершенно новых функций следует начать с подготовки их основы. В этой главе сначала будет рассмотрено построение форм, а затем их использование для аутентификации пользователя и создания нового содержимого.

Имейте в виду, в этой главе мы познакомимся с различными способами построения форм. Все они являются допустимыми, и их выбор зависит от целей применения, определяемых требованиями конкретного проекта.

В этой главе рассматриваются следующие темы:

- создание интерактивных элементов ввода в формах с помощью директив;
- поддержка двусторонней привязки данных в Angular 2;
- привязка моделей данных и интерфейсов к формам и элементам ввода;
- декларативный и императивный способы создания элементов управления;
- альтернативные способы проверки допустимости ввода;

- разработка собственных функций проверки допустимости (валидаторов);
- создание собственной формы аутентификации пользователя;
- реализация поставщиков авторизации общего назначения;
- создание защищенных областей сайта, требующих аутентификации пользователя.

Двухсторонняя привязка данных в Angular 2

Как уже упоминалось в предыдущих главах, одним из основных отличий Angular 2 от предыдущих версий фреймворка является отказ от двусторонней привязки данных как основного шаблона управления данными. Это не совсем верно. Хотя многие из процессов управления данными в Angular 2 являются односторонними, в управлении формами оставлена ниша для двусторонней привязки данных с помощью директивы `NgModel`.

Рассмотрим это на реальном примере. В предыдущей главе был добавлен новый компонент, позволяющий расширить диапазон компонентов, доступных в приложении, чтобы получить больше возможностей для навигации по сайту и заодно исследовать реализацию маршрутизатора. Этот новый компонент – `TaskEditorComponent` – пока не был реализован, его шаблон приводится ниже:

APP/TASKS/TASK-EDITOR.COMPONENT.HTML

```
<form class="container">
  <h3>Task Editor:</h3>
  <div class="form-group">
    <input type="text"
    class="form-control"
    placeholder="Task name"
    required>
  </div>

  <div class="form-group">
    <input type="Date"
    class="form-control"
    required>
  </div>

  <div class="form-group">
    <input type="number"
```

```
    class="form-control"
    placeholder="Pomodoros required"
    min="1"
    max="4"
    required>
  </div>

  <div class="form-group">
    <input type="checkbox" name="queued">
    <label for="queued"> this task by default?</label>
  </div>

  <p>
    <input type="submit" class="btn btn-success" value="Save">
    <a [routerLink]="['TasksComponent']" class="btn btn-danger">
      Cancel
    </a>
  </p>
</form>
```

Это хоть и маленькая, но законченная веб-форма. Компонент включает обработку нескольких событий жизненного цикла маршрутизатора, служащих подтверждением прохождения разных этапов при продвижении по конвейеру навигации. Кроме того, безжизненная форма подобна застывшему на фотографии движению.



В различных примерах к этой главе упоминаются классы, используемые для стилизации формы. Эти имена классов, если не указано иное, заимствованы из таблицы стилей Bootstrap, например `container`, `form-group`, `form-control` и др. Angular никак не связан с ними, и их применение при использовании фреймворка необязательно.

Директива NgModel

Вдохнем в нее жизнь! Одним из преимуществ поддержки двунаправленной привязки данных элементов форм является отсутствие необходимости что-либо импортировать. Angular 2 достаточно умен, чтобы найти то, что ему нужно, а единственная директива, которая нам потребуется, уже доступна по умолчанию. Имеется в виду директива `NgModel`. Согласно официальной документации Angular 2:

> «*Директива ngModel связывает существующую предметную модель с элементами формы. Для двусторонней привязки следует использовать [(ngModel)], что обеспечит обновление модели в обоих направлениях*».

То есть после добавления атрибута ngModel в элемент формы этот элемент начнет отслеживать значение указанного свойства компонента, с которым он связан, и будет обновляться при изменении значения в модели. Вы можете подумать, что это уже делалось в Angular и раньше, причем не сопровождалось фанфарами. Да, но главное отличие состоит в том, что теперь слежение ведется в обоих направлениях. Это значит, что модель будет обновлять свое состояние сразу после изменения значения в элементе формы.

Но достаточно разговоров, пора действовать! Изменим компонент редактирования заданий, чтобы проверить все это на практике. Взгляните еще раз на код компонента и обратите внимание на декоратор CanActivate, задающий пользователю вопрос о переходе. Удалим его, поскольку в этой главе мы увидим более безопасные и элегантные способы, обеспечивающие ту же функциональность. А теперь добавим новое свойство taskName, представляющее имя задания!

APP/TASKS/TASK-EDITOR.COMPONENT.TS

```
export default class TaskEditorComponent implements
OnActivate, CanDeactivate, OnDeactivate {
  taskName: string;

  constructor(private title: Title) {}

  // Остальная часть компонента без изменений
}
```

Откройте соответствующий шаблон и внесите следующие изменения в первый элемент ввода. Они объясняются чуть ниже:

APP/TASKS/TASK-EDITOR.COMPONENT.HTML

```
<p>Your task name is {{taskName}}</p>
<div class="form-group">
  <input type="text"
    class="form-control"
    placeholder="Task name"
    [(ngModel)]="taskName">
</div>
```

Как видите, в элемент ввода добавлен атрибут с директивой [(ngModel)], указывающий на строковое свойство компонента, значение которого дополнительно выводится на экран непосредственно над элементом ввода. Выполните код и измените значение в поле ввода. Вы увидите, как вводимый текст немедленно отображается на экране.

Синтаксис ngModel очень хорошо все это иллюстрирует. В одном и том же атрибуте сочетаются обработчик событий и привязка свойства (о чем свидетельствует заключение в круглые и квадратные скобки), что позволяет внедрить значение в целевой элемент и одновременно отслеживать его изменение. Другими словами, это – двусторонняя привязка данных.

Очевидно, что это весьма упрощенный пример, и обычно требуется создавать нечто более амбициозное, поэтому используем этот вновь приобретенный навык для разработки чего-то более полезного. В следующем разделе мы:

- свяжем форму с вновь созданным экземпляром объекта задания, выступающего в качестве модели;
- заполним модель значениями, введенными в форме;
- сохраним изменения после проверки введенных данных;
- вернем пользователя на страницу со списком заданий, чтобы он мог увидеть вновь созданное задание.

Типы привязки к форме с помощью директивы NgModel

Удалим только что добавленный код и импортируем в форму интерфейс Task вместе с диспетчером TaskService, добавив следующую инструкцию импорта в начало:

APP/TASKS/TASK-EDITOR.COMPONENT.TS

```
import { Component } from '@angular/core';
import { Title } from '@angular/platform-browser';
import {
  Router,
  ROUTER_DIRECTIVES,
  ComponentInstruction,
  CanActivate,
  OnActivate,
  CanDeactivate,
  OnDeactivate } from '@angular/router-deprecated';
import {
  Task,
  TaskService } from '../shared/shared';
```

Обратите внимание, что также был импортирован тип Router из angular2/router. Он необходим, чтобы вернуть пользователя на страницу со списком после успешного создания каждого нового задания.

А теперь добавим новое свойство класса, аннотированное типом `Task`, и объявим зависимость `TaskService` в конструкторе, чтобы иметь возможность создавать новые задания. Удалите созданное ранее строковое поле `taskName` и измените класс, как показано ниже:

APP/TASKS/TASK-EDITOR.COMPONENT.TS

```
export default class TaskEditorComponent implements
OnActivate, CanDeactivate, OnDeactivate {
  task: Task;

  constructor(
    private title: Title,
    private router: Router,
  private taskService: TaskService) {
      this.task = <Task>{};
  }

  // Остальная часть компонента без изменений
}
```

Мы добавили в класс новое поле, представляющее модель `Task`, с которой будет связана форма. Поскольку модель `Task` имеет тип интерфейса, ее экземпляры нельзя создавать с помощью ключевого слова `new`, так как интерфейсы не имеют конструкторов. Однако можно воспользоваться преимуществами обобщения и привести пустой объект к типу интерфейса `Task`, как это и сделано в предыдущем коде.

С другой стороны, объявление типов в конструкторе гарантирует, что инструмент внедрения зависимостей Angular сделает соответствующие поля доступными другим членам компонента сразу после создания его экземпляра.

В идеале достаточно связать данные формы с объектом, в свойстве `task` компонента, сохраняя его во всем приложении с помощью методов, уже созданных в классе `TaskService`, а затем сразу перейти к списку заданий. Начнем с внесения изменений в HTML-шаблон, куда добавим необходимые директивы `ngModel`, а также кнопку **Submit** и обработчик события отправки формы, обернув их в тег формы:

APP/TASKS/TASK-EDITOR.COMPONENT.HTML

```
<form class="container" (submit)="saveTask()">
  <h3>Task Editor:</h3>
  <div class="form-group">
    <input type="text"
      class="form-control"
```

```
    placeholder="Task name"
    [(ngModel)]="task.name">
</div>

<div class="form-group">
  <input type="date"
    class="form-control"
    [(ngModel)]="task.deadline">
</div>

<div class="form-group">
  <input type="number"
    class="form-control"
    placeholder="Pomodoros required"
    min="1"
    max="4"
    [(ngModel)]="task.pomodorosRequired">
</div>

<div class="form-group">
  <input type="checkbox"
    name="queued"
    [(ngModel)]="task.queued">
      <label for="queued"> this task by default?</label>
  </div>

  <p>
    <input type="submit" class="btn btn-success" value="Save">
    <a [routerLink]="['TaskList']" class="btn btn-danger">
      Cancel
    </a>
  </p>
</form>
```

В этом фрагменте следует отметить два примечательных момента:

- каждый элемент ввода, включающий директиву `ngModel`, сопоставляется одному из свойств типа `Task`, экземпляр которого хранится в свойстве `task` контроллера;
- почему в форму включена кнопка **Submit**, хотя тег `form` не имеет атрибута `action`, определяющего адрес отправки формы? Об этом позаботится обработчик события `(submit)`.

В форме определены три поля ввода, использующих механизм двухсторонней привязки. Каждое из них связывается со свойством в объекте модели. Когда осуществляется отправка формы, вызывается метод `saveTask()`, размещенный в теле компонента. Рассмотрим

его. Он еще не был добавлен в класс, поэтому добавьте в компонент следующий метод, разместив его где угодно, но после конструктора:

APP/TASKS/TASK-EDITOR.COMPONENT.TS

```
saveTask() {
  this.task.deadline = new Date(this.task.deadline.toString());
  this.taskService.addTask(this.task);
  this.router.navigate(['TaskList']);
}
```



Вас, вероятно, удивила первая строка в методе `saveTask()`. Да, выглядит странно. В ней значение свойства `deadline` (объект `Date`) преобразуется в строку, а затем снова преобразуется в объект `Date`. На это есть веская причина. На момент написания книги фильтр `DatePipe` (подобный тому, что используется в шаблоне `TasksComponent`) принимает только объекты `Date`, отформатированные без учета региональных настроек. Поле ввода `date` не поддерживает региональных настроек, поэтому для согласования данных нужно перед их сохранением повторно выполнить их форматирование. Существуют и другие, более удачные решения, но они, как правило, более громоздки и не имеют отношения к рассматриваемой теме, поэтому будем придерживаться этого подхода и в остальной части этой главы.

Обход события CanDeactivate маршрутизатора при отправке формы

Теперь компонент обладает всем необходимым для отражения изменений в модели посредством формы. Но если пользователь попытается заполнить форму сведениями о новом задании и сохранить его, он столкнется с докучливым диалогом, созданным в предыдущей главе и приглашающим заполнить форму, то есть сделать именно то, что только что было сделано! Исправим это прямо сейчас. Добавим переменную, информирующую о том, была форма обновлена и успешно сохранена или нет, и используем ее, чтобы пропустить вывод диалога при необходимости:

APP/TASKS/TASK-EDITOR.COMPONENT.TS

```
export default class TaskEditorComponent implements
OnActivate, CanDeactivate, OnDeactivate {
  task: Task;
  changesSaved: boolean;

  constructor(
```

```
    private title: Title,
    private router: Router,
    private taskService: TaskService) {
      this.task = <Task>{};
  }

  saveTask() {
    this.task.deadline = new Date(this.task.deadline.toString());
    this.taskService.addTask(this.task);
    this.changesSaved = true;
    this.router.navigate(['TaskList']);
  }

  routerOnActivate() {
    this.title.setTitle('Welcome to the Task Form!');
  }

  routerCanDeactivate() {
    return this.changesSaved ||
      confirm('Are you sure you want to leave?');
  }

  routerOnDeactivate() {
    this.title.setTitle('My Angular 2 Pomodoro Timer');
  }
}
```

По сути, поле `changesSaved` – это логический флажок, принимающий значение `true` сразу после сохранения задания `Task` в методе `saveTask`. Он позволяет методу `routerCanDeactivate()` либо просто вернуть значение `true`, если изменения были сохранены, либо вывести диалог подтверждения.

Пока все идет хорошо. А теперь проявим творческий подход и немного улучшим логику формы. Проверка значений полей ввода перед отправкой – отличная отправная точка, ведущая к следующему этапу путешествия по миру форм Angular 2.

Отслеживание изменений и проверка допустимости данных

Мы уже реализовали отслеживание изменений в форме с помощью двусторонней привязки данных, но следует также учитывать общее состояние формы. Для этого можно воспользоваться директивой `NgForm`. Директива `NgForm` отслеживает состояния всех элементов вво-

да, заключенных внутри нее. Самое интересное, что такая директива присутствует в примере изначально. Но как? Обычно директива `NgForm` присоединяется к любому селектору тега `<form>`, присутствующему в шаблоне, предоставляя дополнительные возможности отслеживания состояния полей ввода формы, с учетом проверки допустимости и взаимодействий с пользователем. Другими словами, если в шаблоне есть форма, в нем имеется и директива `ngForm`.



Автоматическое связывание можно отменить, добавив атрибут `ngNoForm` в любой тег `<form>`, где не требуется вмешательство Angular.

Рассмотрим простой пример. Во-первых, добавим во все поля атрибут `required`:

APP/TASKS/TASK-EDITOR.COMPONENT.HTML

```
<form class="container" (submit)="saveTask()">
  <h3>Task Editor:</h3>
  <div class="form-group">
    <input type="text"
      class="form-control"
      placeholder="Task name"
      [(ngModel)]="task.name"
      required>
</div>

<div class="form-group">
  <input type="date"
    class="form-control"
    [(ngModel)]="task.deadline"
    required>
</div>

<div class="form-group">
  <input type="number"
    class="form-control"
    placeholder="Pomodoros required"
    min="1"
    max="4"
    [(ngModel)]="task.pomodorosRequired"
    required>
</div>

<div class="form-group">
  <input type="checkbox"
    name="queued"
```

```
    [(ngModel)]="task.queued">
    <label for="queued"> this task by default?</label>
</div>

<p>
  <input type="submit" class="btn btn-success" value="Save">
  <a [routerLink]="['TaskList']" class="btn btn-danger">
    Cancel
  </a>
  </p>
</form>
```

Если теперь пытаться отправить форму, браузер автоматически выведет предупреждение, сгенерированное его механизмом проверки допустимости, поддержка которого регламентируется стандартом HTML5. Но основная работа при этом происходит за кулисами. Просматривая код формы с помощью инструментов разработчика в браузере, можно заметить множество имен классов, добавленных в элементы ввода. Откуда все это взялось? Ответ прост! Все эти имена добавила директива `NgForm`, присоединенная к тегу `<form>`. Рассмотрим для примера первое поле ввода:

```
<input type="text"
  class="form-control ng-untouched ng-pristine ng-invalid"
  placeholder="Task name"
  required="">
```

Имена классов (их легко идентифицировать по префиксу `ng-`) подсказывают различные состояния, которые получают любые элементы ввода внутри директивы `NgForm`. Эти классы обеспечивают реактивность изменений в полях ввода. Оставив панель инструментов разработчика открытой, выберите любое поле ввода, измените его значение, а затем очистите, чтобы пронаблюдать смену классов для следующих состояний:

- **Untouched**: если имеет значение `true`, пользователь не воздействовал на элемент;
- **Touched**: если имеет значение `true`, пользователь воздействовал на элемент;
- **Pristine**: элемент и соответствующая ему модель не изменялись;
- **Dirty**: элемент и соответствующая ему модель были изменены;
- **Valid**: внутренняя модель является допустимой;
- **Invalid**: внутренняя модель не является допустимой.

В процессе взаимодействий с элементами формы вы увидите сгенерированные имена классов, соответствующие этим состояниям, с префиксом ng-, например: ng-untouched, ng-pristine, ng-invalid и т. д.

Отслеживание изменений с помощью локальных ссылок

Теперь, когда известно, как элементы ввода могут реагировать на действия пользователя и проверку допустимости модели, можно перейти к следующему этапу и заняться выводом на экран дополнительной информации. Например, можно применить к форме стили в реактивной манере:

APP/TASKS/TASK-EDITOR.COMPONENT.TS

```
@Component({
  selector: 'pomodoro-tasks-editor',
  directives: [ROUTER_DIRECTIVES],
  providers: [Title],
  templateUrl: 'app/tasks/task-editor.component.html',
  styles: [`
    .ng-valid { border-color: #3c763d; }
    .ng-invalid { border-color: #a94442; }
    .ng-untouched { border-color: #999999; }
  `]
})
...
```

Теперь форма будет выводить визуальную подсказку о состоянии каждого элемента ввода через макет отображения. Однако вывод дополнительных сообщений на экран мог бы сделать приложение более удобным для пользователя. Для этого необходимо определить состояние каждого элемента ввода, для чего удобно использовать локальные ссылки в шаблоне. Они открывают доступ к универсальному обработчику состояния формы, которым является директива ngForm. Добавим сообщение о состоянии в форму и реализуем слежение за состоянием первого поля ввода, используя директиву ngForm в качестве посредника:

APP/TASKS/TASK-EDITOR.COMPONENT.HTML

```
<form class="container" (submit)="saveTask()">
  <h3>Task Editor:</h3>
  <div class="form-group">
    <p class="text-muted" *ngIf="name.untouched">
      Start here by entering the task name.
```

```
</p>
<p class="text-success" *ngIf="name.valid && name.touched">
  Well done! That's a good name for a task!
</p>
<p class="text-danger" *ngIf="!name.valid && name.touched">
  Oops! You cannot leave the name blank...
</p>
<input type="text"
       class="form-control"
       placeholder="Task name"
       [(ngModel)]="task.name"
       #name="ngForm"
       required>
  </div>
...
```

Рассмотрим этот фрагмент. Первый блок просто отображает различные сообщения (с использованием стилей Bootstrap, которые уже применялись к остальной части формы), в зависимости от состояния элемента ввода. Для ссылки на элемент ввода из других элементов необходима локальная переменная шаблона, поэтому в элемент ввода была добавлена локальная ссылка `#name`. Удивление вызывает присваиваемое значение. Локальным переменным шаблонов в Angular 2 можно присваивать любые значения или указатели на другие объекты, именно поэтому переменной `#name` присваивается строка `"ngForm"`. Директива `NgForm` экспортирует себя под именем `ngForm`, что позволяет, сославшись на ее имя в локальной ссылке шаблона, получить доступ к ее программному интерфейсу, а следовательно, и к ее состоянию.

А теперь отвлечемся от построения форм, основанных на классической двусторонней привязке данных. Существует еще один мощный способ создания форм в Angular 2, не имеющий ничего общего с директивой `[(ngModel)]`, но тем не менее обеспечивающий поддержку тех же и даже нескольких дополнительных функций, что делает его более предпочтительным.

Элементы управления, группы элементов управления и класс FormBuilder

В этой главе мы узнали, как реализовать двустороннюю привязку для подключения сущностей данных к полям ввода в формах. Это достаточно совершенная методика, но Angular 2 предлагает еще более эффективную модель, где все потоки данных однонаправленные. Этому

существует много оправданий, но наиболее весомой причиной является более низкая производительность традиционных двусторонних привязок данных, по сравнению с другими моделями, где информация перемещается только в одном направлении.

Объекты Control и Validator

Объекты `Control` можно кратко охарактеризовать так: объект `Control` – это минимальное представление элемента управления, являющегося частью формы. Идеальными примерами элементов управления могут служить элементы ввода текста или флажки. По сути, элементы управления Angular 2 использовались на протяжении всей этой главы, каждый раз, когда в форму создания Pomodoro-задания добавлялись поля ввода. Фреймворк Angular 2 (по умолчанию) создает элемент управления для каждого компонента формы, помещенного внутрь директивы `NgForm`. Поскольку элементы формы присоединены к этой директиве, мы изначально работали с объектами `Control`.

Итак, что нужно сделать, чтобы создать объект `Control`? В действительности все довольно просто. Импортировав тип `Control` из барреля `angular2/common`, элементы управления можно создавать, как показано ниже:

```
var firstName = new Control('', Validators.required);
```

Конструктор класса `Control` весьма прост: первый параметр – это значение по умолчанию элемента управления, может быть объектом любого типа. Обычно в этом параметре передается пустая строка, когда значение по умолчанию не требуется. Во втором параметре должна передаваться функция-валидатор, используемая объектом `Control` для проверки допустимости введенных данных. Как будет показано ниже в этой главе, имеется возможность создавать свои валидаторы или пользоваться статическими функциями из класса `Validator`, также входящего в состав барреля `angular2/common`. Этот класс имеет следующие статические методы-валидаторы:

- `required`: требует, чтобы элемент управления имел непустое значение; эквивалентен атрибуту `required` в HTML5;
- `minLength(minLength: number)`: ограничивает минимальную длину значения элемента управления;
- `maxLength(maxLength: number)`: ограничивает максимальную длину значения элемента управления;
- `pattern(pattern: string)`: проверяет значение на соответствие регулярному выражению в аргументе `pattern`;

- `compose(validators: Function[])`: в действительности является не методом, а примесью, объединяющей валидаторы из массива, передаваемого в параметре;
- `composeAsync()`: то же, что и `compose`, но требует преобразования каждой функции в объект `Promise` и выполняет их все сразу, возвращая результат проверки после разрешения всех объектов `Promise`.

Каждый из этих валидаторов возвращает либо значение `null`, указывающее, что входные данные являются допустимыми, либо объект с подробным описанием причин, почему содержимое элемента управления не может быть признано допустимым. Не волнуйтесь! Далее вы получите полное представление, когда мы реализуем несколько таких валидаторов в реальном примере.

Элементы управления в модели DOM и директива ngControl

Объекты `Control` можно создавать в теле класса компонента или поручить их создание фреймворку Angular 2, внедрив их непосредственно в шаблоны с помощью директивы `ngControl`, формализуемой в классе `NgControlName`.



Не путайте класс `NgControlName` с классом `NgControl`, который является базовым для класса `NgControlName`.

Эта директива может использоваться только как дочерняя по отношению к директиве `NgForm` (была рассмотрена выше) или элементу с директивой `NgFormModel`, которая рассматривается ниже. Официальная документация Angular описывает директиву `ngControl` так:

> *«Создает и привязывает элемент управления с указанным именем к элементу DOM. Эта директива может использоваться только как дочерняя по отношению к NgForm или NgFormModel».*

В соответствии с этой цитатой директива `ngControl` обычно используется для декорирования элементов ввода, как показано ниже:

```
<input type="email" ngControl="email">
```

Добавив директиву `NgControlName` в элементы ввода, мы получаем возможность читать их значения, проверять допустимость и реагировать на изменения в их состоянии. Проще всего для этой цели использовать локальные ссылки, указывающие на директиву `NgForm`:

```
<input type="password" ngControl="password" #pwd="ngForm">
<div *ngIf="!pwd.valid">Password is invalid</div>
```

В предыдущем примере, который предназначен для работы в контексте элемента формы, элемент ввода с уже имеющейся директивой `ngControl` отмечается локальной ссылкой `#pwd`, указывающей на `ngForm`. Обнаружив эту ссылку, фреймворк Angular установит указатель на сам элемент управления. В результате мы получаем удобный инструмент для доступа и проверки свойства объекта `Control`, присоединенного к директиве `ngControl`. Этот прием часто применяется при работе с формами в Angular 2.



Стоит отметить, что статические методы класса `Validator` имеют аналоги в форме директив, которые можно применять к элементам с директивой `ngControl` для проверки допустимости:

```
<input ngControl="taskName" required pattern="[a-zA-Z]*">
<input ngControl="pomodoros" minlength="1" maxlength="5">
```

Группировка элементов в модели DOM с помощью директивы NgControlGroup

Обычно формы включают несколько элементов ввода, и поэтому модель данных, которую они представляют, имеет несколько полей или свойств. Определяя набор входных данных, можно воспользоваться директивой `NgControlGroup`. Эта директива действует подобно ссылке, обертывающей несколько элементов ввода с директивой `ngControl`, которую можно использовать как единую точку входа для проверки значений различных элементов ввода, включенных в `NgControlGroup`. В следующем примере определяется простая форма с двумя полями ввода, завернутыми в группу `NgControlGroup`, ссылка на которую передается в метод компонента при отправке формы:

```
import { Component } from '@angular/core';
import { NgControlGroup } from '@angular/common';

@Component({
  selector: 'hello-form',
  template: `
    <form (submit)="sayHello(fullName)">
      <div ngControlGroup="nameControlsGroup" #fullName="ngForm">
        <div class="form-group">
          <input type="text"
            placeholder="First name"
```

```
        ngControl="firstName"
        required>
      <input type="text"
        placeholder="Last name"
        ngControl="lastName"
        required>
    </div>
  </div>
    <input type="submit" value="Say my name">
  </form>
  `
})
export default class SayHello {
  sayHello(controlGroup: NgControlGroup): void {
    if (controlGroup.control) {
      let firstName = controlGroup.control.value.firstName;
      let lastName = controlGroup.control.value.lastName;
      alert(`Hello ${firstName} ${lastName}!`);
    }
  }
}
```

Здесь локальные ссылки используются для анализа в `NgControlGroup` с применением `ngForm` в качестве посредника. Объект, переданный в метод `sayHello()`, является ссылкой на группу `NgControlGroup` и экспортирует (через свойство `control`) всю информацию о глобальном состоянии группы: отсутствие изменений, допустимость, наличие изменений и т. д. Имеется также возможность просмотра объектов `NgControl`, содержащихся в группе элементов, через внутреннее свойство `controls`. Свойство `value` элементов возвращает объект хэша, представляющий значения всех полей ввода.

В этом примере директиве `NgControlGroup` присвоено значение `nameControlsGroup`. Это строковое значение становится именем группы элементов в контексте обертывающей директивы `NgForm`. Если проследовать по локальной ссылке к элементу `<form>` и просмотреть ее свойство `controls`, вы обнаружите в нем объект, содержащий свойство с именем группы (или групп), ссылающийся на соответствующий объект `NgControlGroup`. С другой стороны, заглянув в свойство `value`, вы обнаружите объект хэша со множеством свойств, соответствующих группам элементов управления в форме, содержащих представления значений элементов управления в виде объектов.

Императивное определение групп элементов управления с помощью ControlGroup

В начале главы был описан процесс императивного объявления элементов управления в классе контроллера компонента. Затем речь зашла о модели DOM и создании элементов управления непосредственно в шаблонах, а также о создании подмножеств элементов, сгруппированных с помощью директивы `NgControlGroup`.

Группировка элементов управления является распространенной операцией, и ее можно выполнять в классе контроллера компонента, создавая экземпляры объектов `ControlGroup` с помощью класса `FormBuilder`, генерирующего объекты форм (иными словами, группы элементов управления) путем анализа конфигурации отобранных полей ввода. Рассмотрим пример класса компонента, который после внедрения `FormBuilder` через конструктор использует его методы для создания экземпляра объекта `ControlGroup` с вложенными объектами `ControlGroup` и `Controls`:

```
import { Component } from '@angular/core';
import {
  Control,
  ControlGroup,
  FormBuilder,
  Validators } from '@angular/common';

@Component({
  selector: 'my-login',
  providers: [FormBuilder],
  templateUrl: 'my-login.component.html'
})
export class LoginComponent {
  name: Control;
  username: Control;
  password: Control;
  signupForm: ControlGroup;

  constructor(formBuilder: FormBuilder) {
    this.name = new Control('', Validators.required);
    this.username = new Control('', Validators.required);
    this.password = new Control('', Validators.required);

    this.signupForm = formBuilder.group({
      name: this.name,
      credentials: formBuilder.group({
```

```
        username: this.username,
        password: this.password
      })
    });
  }
}
```

Предыдущий пример создает компонент с объектом `ControlGroup`, представляющим типичную форму аутентификации с полем `signupForm` класса `LoginComponent`. Это поле заполняется в методе `formBuilder.group()` элементами управления `name` и `credential`, последний из которых является группой (содержащей два элемента управления), с разными видами проверки допустимости. Каждый из этих элементов управления, по сути, является экземпляром класса `Control`. Для создания экземпляров объекта `Control` можно также использовать метод `control()` класса `FormBuilder` и тем самым избавиться от необходимости импортировать тип `Control`, если он нигде больше не используется. Это позволяет значительно упростить реализацию предыдущего примера:

```
export class LoginComponent {
  signupForm: ControlGroup;

  constructor(formBuilder: FormBuilder) {
    this.signupForm = formBuilder.group({
      name: this.formBuilder.control('', Validators.required]),
      credentials: formBuilder.group({
        username: this.formBuilder.control('',
        Validators.required),
        password: this.formBuilder.control('',
        Validators.required)
      })
    });
  }
}
```

Можно пойти еще дальше и воспользоваться удобной синтаксической конструкцией из Angular для создания элементов управления внутри `ControlGroup`:

```
export class LoginComponent {
  signupForm: ControlGroup;

  constructor(formBuilder: FormBuilder) {
    this.signupForm = formBuilder.group({
```

```
      name: ['', Validators.required],
      credentials: formBuilder.group({
        username: ['', Validators.required],
        password: ['', Validators.required]
      })
    });
  }
}
```

Это наиболее простой способ создания экземпляров элементов управления, каждому из которых соответствует объект хэша со свойством `value`, где хранится массив. Первый элемент этого массива содержит значение по умолчанию для данного элемента, а остальные – ссылки на функции-валидаторы, применяемые к элементу.

Эти три способа группировки элементов управления, положенные в основу предполагаемой формы аутентификации, обеспечивают большую гибкость при создании сложных форм из контроллера компонента. Но какой синтаксис предпочтительнее? Это зависит от многих факторов. Последний способ создания экземпляров `ControlGroup` и объектов `Control` является одним из самых популярных из-за своей простоты, предыдущий дает возможность ссылаться на элементы управления из других мест в классе контроллера, учитывая, что такие элементы управления являются частью программного интерфейса. В конечном счете все зависит от того, где должны находиться элементы управления и откуда они должны быть доступны.

Соединение модели DOM и контроллера с помощью ngFormModel

Итак, имеется возможность создавать формы императивно из контроллера компонента. Что дальше? Нужно каким-то образом привязать эту логику к HTML-шаблону, и эту возможность дает нам новая директива `NgFormModel`. Эта директива должна быть привязана к элементу DOM (обычно к форме или любому элементу, содержащему `NgForm`), и в ней должно быть указано имя соответствующего объекта `ControlGroup`. С этого момента объекты `Control`, включенные в объект `ControlGroup`, привязываются к элементам ввода, отмеченным директивой `ngControl`, с соответствующими именами:

```
<form [ngFormModel]="signupForm" (submit)="doSomething($event)">
  <div>
    <input type="text"
```

```
    placeholder="Your name"
    ngControl="name">
</div>

<div ngControlGroup="credentials">
  <input type="text"
    placeholder="Your username"
    ngControl="username">
  <input type="password"
    placeholder="Your password"
    ngControl="password">
  </div>

  <p>
    <input type="submit" value="Signup">
  </p>
</form>
```

Теперь объект `ControlGroup` связан с формой `signupForm`, элементы ввода которой обрабатываются директивой `ngControl`, автоматически отображающей элементы управления `ControlGroup`.

А теперь пришло время воплотить все это в проекте Pomodoro, так что засучим рукава в предвкушении работы, которую нужно будет проделать.

Пример компонента входа пользователя

Ранее в этой главе была реализована форма для создания новых заданий. В действующем приложении эта возможность не должна быть доступна кому угодно, поэтому нужно защитить ее паролем. Создание формы входа станет первым действием в этом разделе, который посвящен разработке функции входа пользователя в систему.

Контекст функции входа

Функция входа в систему существует параллельно с остальными функциональными возможностями приложения и поэтому заслуживает отдельной папки и фасада. Вся реализация будет зависеть от компонента `LoginComponent`, поэтому начнем с создания основных файлов в папке для новой функции входа в систему, расположенной на том же уровне, что и папки `shared`, `tasks` и `timer`. Сначала создадим компонент без реализации с соответствующим шаблоном и фасадом, а о деталях реализации позаботимся позднее.

app/login/login.component.ts – класс контроллера компонента

```
import { Component } from '@angular/core';
import {
  FormBuilder,
  ControlGroup,
  Validators,
  Control } from '@angular/common';
import { Router } from '@angular/router-deprecated';

@Component({
  selector: 'pomodoro-login',
  templateUrl: 'app/login/login.component.html'
})
export default class LoginComponent {
  loginForm: ControlGroup;
  notValidCredentials: boolean = false;

  constructor(
    formBuilder: FormBuilder,
    private router: Router) {}

    authenticate() {}
}
```

app/login/login.component.html – шаблон компонента

```
<form [ngFormModel]="loginForm"
  class="container"
  (ngSubmit)="authenticate()">

  <h3>This content is password-protected</h3>
  <p>Please enter your credentials below</p>

  <div class="alert alert-danger" *ngIf="notValidCredentials">
    Your credentials are not valid!
  </div>

  <div class="form-group">
    <input type="text"
      class="form-control"
      placeholder="Your username"
      ngControl="username">
</div>

<div class="form-group">
  <input type="password"
    class="form-control"
    placeholder="Your password"
    ngControl="password">
```

```
  </div>

  <p>
    <input type="submit"
      class="btn btn-success"
      value="Authenticate"
      [disabled]="!loginForm.valid">
  </p>
</form>
```

APP/LOGIN/LOGIN.TS – фасад функции

```
import LoginComponent from './login.component';

export {
  LoginComponent
};
```

Дополнения в конфигурации маршрутизатора корневого компонента очень помогут при тестировании изменений, которые будут вноситься на протяжении нескольких следующих страниц. Для этого отредактируем различные части корневого компонента, как показано ниже:

APP/APP.COMPONENT.TS – блок операторов импорта

```
import { Component } from '@angular/core';
import { SHARED_PROVIDERS } from './shared/shared';
import { HTTP_PROVIDERS } from '@angular/http';
import { ROUTER_PROVIDERS, RouteConfig, ROUTER_DIRECTIVES, Router }
from '@angular/router-deprecated';
import { TimerComponent } from './timer/timer';
import { TasksComponent, TaskEditorComponent } from './tasks/tasks';
import { FORM_PROVIDERS } from '@angular/common';
import { LoginComponent } from './login/login';
...
```

APP/APP.COMPONENT.TS – параметры RouteConfig

```
...
@RouteConfig([
  { path: '',
    name: 'Home',
    redirectTo: ['TasksComponent']
  }, {
    path: 'tasks',
    name: 'TasksComponent',
    component: TasksComponent,
    useAsDefault: true
```

```
  }, {
    path: 'tasks/editor',
    name: 'TaskEditorComponent',
    component: TaskEditorComponent
  }, {
    path: 'timer/...',
    name: 'TimerComponent',
    component: TimerComponent
  }, {
    path: 'login',
    name: 'LoginComponent',
    component: LoginComponent
  }
])
export default class AppComponent {}
```

Шаблон формы входа

Достаточно лишь бросить взгляд на форму, чтобы понять механизм, который следует построить в классе контроллера компонента, чтобы вдохнуть в нее жизнь. Директива `NgformModel` указывает на объект `ControlGroup` с именем `loginForm`, который еще не создан. Объект `ControlGroup` послужит оберткой для двух объектов `Control` (имя пользователя и пароль), а его состояние будет определять доступность кнопки **Submit** в нижней части формы. Компонент будет отличаться от прочих наличием метода `authenticate()`, обрабатывающего событие отправки формы. И последнее, но важное замечание: непосредственно перед элементами ввода имеется маленький фрагмент кода:

```
<div class="alert alert-danger" *ngIf="notValidCredentials">
  Your credentials are not valid!
</div>
```

Как вы уже догадались, это сообщение будет отображаться, если введены неверные учетные данные. Проверка выполняется в методе `authenticate()`, как показано ниже. В примере присутствует новая директива события `ngSubmit`. Согласно документации Angular, она оповещает об отправке формы. Использование этой директивы заменяет привязку события `(submit)`, которая использовалась до сих пор.

Компонент входа

Теперь определим реализацию класса компонента входа. У нас уже имеется каркас класса контроллера, но его нужно оснастить конструктором и методом `authenticate()`:

APP/LOGIN/LOGIN.COMPONENT.TS

```
...
@Component({
  selector: 'pomodoro-login',
  templateUrl: 'app/login/login.component.html'
})
export default class LoginComponent {
  loginForm: ControlGroup;
  notValidCredentials: boolean = false;

  constructor(
    formBuilder: FormBuilder,
    private router: Router) {
      this.loginForm = formBuilder.group({
      username: ['', Validators.required],
      password: ['', Validators.required]

    });
  }

  authenticate() {
    let credentials: any = this.loginForm.value;
      this.notValidCredentials = !this.loginForm.valid &&
                                 this.loginForm.dirty;

      if(credentials.username === 'john.doe@mail.com' &&
         credentials.password === 'letmein')
      {
        this.router.navigateByUrl('/');
      } else {
        this.notValidCredentials = true;
      }
  }
}
```

Этот фрагмент здорово напоминает все, что мы видели в предыдущих разделах. Здесь для краткости пропущены инструкции импорта, но и так ясно, что с помощью приема внедрения зависимостей через конструктор, которому отдается предпочтение в Angular 2, в класс внедряются зависимости `Router` и `FormBuilder` (а также конфигурация объекта маршрутизатора как закрытое свойство). С помощью экземпляра `FormBuilder` создается требуемый объект `ControlGroup`, присваивается свойству `loginForm` и позднее привязывается к директиве `NgFormModel`, декорированной директивой `NgForm` (помните, что это элемент `form`?), необходимой шаблону.

Вызовом статического метода `Validators.required` оба элемента ввода объявляются обязательными. Однако реализация метода `authenticate()` требует особого внимания. Разберем ее по частям:

```
let credentials: any = this.loginForm.value;
this.notValidCredentials = !this.loginForm.valid &&
                           this.loginForm.dirty;
```

Метод `authenticate()` сначала связывает свойство `value` группы элементов `loginForm` с переменной `credentials`. Как вы уже знаете, `loginForm.value` хранит объект хэша следующего вида:

```
{ username: "", password: "" }
```

Затем проверяются допустимость и состояние группы элементов управления `loginForm`, и результат присваивается логическому свойству `notValidCredentials`. И наконец, последний фрагмент просто проверяет правильность введенных учетных данных. В случае успешной проверки пользователь отправляется на начальную страницу (можете выбрать любую другую страницу, более уместную на ваш взгляд), а при вводе неправильных учетных данных на экран выводится предупреждение.



Для простоты учетные данные сравниваются с фиксированными значениями, но вы никогда не должны, и главным здесь является слово *никогда*, применять этот подход при проверке регистрационной информации для предоставления доступа к деликатным частям приложения. Он слишком опасен, и злобным хакерам не составит труда взломать такой сайт. На последующих страницах также будет затронута тема проверки подлинности, но и там предпочтение будет отдаваться простоте примеров. В реальной ситуации *всегда* следует полагаться на удаленный программный интерфейс аутентификации и зашифрованные маркеры для сохранения сеанса. Эти проблемы ближе к серверному программированию, и их обсуждение выходит за рамки этой книги.

Применение пользовательских проверок допустимости к элементам управления

Мы уже видели, как проверять допустимость элементов управления, добавляя готовые валидаторы при их создании, но нам еще нужно получить ответ на два связанных с этим вопроса:

- Как создавать собственные валидаторы?
- Как указать несколько валидаторов в одном объекте `Control`?

Этот раздел дает ответы на оба вопроса. Вообще говоря, валидатор – это функция, принимающая объект `Control` в аргументе и возвращающая значение `null` (если объект `Control` является допустимым) или объект хэша с парами ключ/значение. В форме входа нам требуется проверить правильность имени пользователя. Поскольку предполагается, что роль имени пользователя будет играть адрес электронной почты, вполне достаточно будет проверить, содержит ли поле ввода имени пользователя правильный адрес электронной почты.



Обычно для этого используется валидатор `pattern`, но целью этого примера является демонстрация создания собственного валидатора.

Итак, двинемся дальше. Создадим функцию проверки адреса электронной почты и добавим ее в тело класса `LoginComponent`:

APP/LOGIN/LOGIN.COMPONENT.TS

```
private emailValidator(control: Control):
{ [key: string]: boolean } {
  if (!/(.+)@(.+){2,}\.(.+){2,}/.test(control.value)) {
    return {
      'emailNotValid': true
    };
  }

  return null;
}
```

Структура возвращаемого объекта при недопустимом значении элемента управления не так проста. Этот объект будет добавлен в свойство `errors` элемента управления, аннотированного функцией `Validator`. Это очень удобно, поскольку позволяет вернуть дополнительные сведения об источнике ошибки при оценке допустимости значения каждого элемента ввода, особенно когда элементы контролируются несколькими адаптерами проверки допустимости.

Кстати, о нескольких адаптерах: нам необходимо включить две проверки в объект `Control`, поскольку требуется проверить ввод с помощью валидатора `required` и вновь созданной функции. Для этого в классе `Validator` имеется статический метод:

```
this.loginForm = formBuilder.group({
  username: ['',
    Validators.compose([
      Validators.required,
```

```
      this.emailValidator
    ])
  ],
  password: ['', Validators.required]
});
```

Отображение изменений состояния в элементах управления

До сих пор рассматривалась только возможность проведения операции в зависимости от изменения значений в элементах ввода, но желательно применить более реактивный подход. Хорошая новость заключается в том, что оба класса – `ControlGroup` и `Control` – имеют по два свойства типа `EventEmitter`, на которые можно подписать собственных наблюдателей. После этого от любого объекта `Control` или обертывающего его объекта `ControlGroup` можно оперативно получать уведомления об изменении его состояния (`pristine`, `touched` и т. д.) или значения. Здесь имеются в виду свойства `statusChanges` и `valueChanges`, которые являются частью любого объекта `Control` или `ControlGroup` и экспортируют тот же интерфейс, что и наблюдаемые объекты HTTP, рассмотренные в *главе 6 «Асинхронная обработка данных в Angular 2»*.

Рассмотрим конкретный пример. Неплохо было бы выводить в режиме реального времени визуальные подсказки при вводе пользователем своего имени, информирующие о его допустимости. Сначала внесем изменения в контроллер компонента, включив в него логическое поле, которое в дальнейшем будет использоваться для включения и выключения визуальных уведомлений в шаблоне:

APP/LOGIN/LOGIN.COMPONENT.TS

```
export default class LoginComponent {
  loginForm: ControlGroup;
  notValidCredentials: boolean = false;
  showUsernameHint: boolean = false;

  constructor(
    formBuilder: FormBuilder,
    private router: Router) {
    this.loginForm = formBuilder.group({
      username: ['', Validators.compose([
        Validators.required,
        this.emailValidator])],
      password: ['', Validators.required]
    });
```

```
  const username = this.loginForm.controls['username'];
  username.valueChanges.subscribe(value => {
  this.showUsernameHint = (username.dirty &&
                           value.indexOf('@') < 0);
});

}
// Остальная часть реализации без изменений...
  ...
}
```

APP/LOGIN/LOGIN.COMPONENT.HTML

```
<div class="form-group">
  <input type="text"
    class="form-control"
    placeholder="Your username"
    ngControl="username">
  <p *ngIf="showUsernameHint"class="help-block">
    That does not look like a proper username
  </p>
</div>
```

Обратите внимание, что ссылка на объект `Control` с именем пользователя осуществляется через свойство `controls` объекта `loginForm`. Имея эту ссылку, можно подписать наблюдателей, отслеживающих изменения в элементе и действующих соответственно этим изменениям. В данном случае будет обновляться значение поля `showUsernameHint`, определяющее необходимость отображения визуальной подсказки.

Фиктивная служба проверки подлинности клиента

Английское слово *mocking* (фиктивный, ненастоящий) довольно широко распространено в модульном тестировании и, пожалуй, лучше всего характеризует то, что нам предстоит создать сейчас. В предыдущем разделе была реализована весьма упрощенная проверка подлинности пользователей, хотя в реальных приложениях забота об этом делегируется отдельной службе, обладающей всеми необходимыми инструментами для организации входа, выхода, а иногда и проверки привилегий пользователя при предоставлении доступа к защищенным областям приложения.

Далее будет создана упрощенная версия такой службы, проверяющей пользователя при входе в систему с помощью компонента, созданного в предыдущем разделе. Кроме того, эта служба обеспечит сохранение маркера подлинности и возможность проверки прав пользователя на доступ к защищенным страницам.

Прежде чем приступать к написанию кода, кратко сформулируем минимальные требования к службе:

- нам нужен метод, обслуживающий вход пользователей;
- также нам потребуется общедоступный метод для выхода из системы;
- третий метод или свойство должно возвращать состояние аутентификации пользователя, чтобы можно было разрешить или запретить переход на защищенные страницы;
- наличие отслеживаемого свойства, информирующего о текущем состоянии активного пользователя применительно к проверке подлинности, позволит сделать пользовательский интерфейс более реактивным.

Основываясь на этих требованиях, приступим к построению идеальной службы аутентификации. Поскольку эта служба не зависит от компонентов и предназначена для всего приложения, поместим ее в папку `services` общего контекста, применив уже знакомые соглашения об именах, и экспортируем ее через общий фасад:

APP/SHARED/SERVICES/AUTHENTICATION.SERVICE.TS

```
import { Injectable, EventEmitter } from '@angular/core';

@Injectable()
export default class AuthenticationService {

  constructor() {}

  login({username, password}): Promise<boolean> {}

  logout(): Promise<boolean> {}

  static isAuthorized(): boolean {}
}
```

APP/SHARED/SHARED.TS

```
import Queueable from './interfaces/queueable';
import Task from './interfaces/task';

import FormattedTimePipe from './pipes/formatted-time.pipe';
import QueuedOnlyPipe from './pipes/queued-only.pipe';
```

```
import AuthenticationService from './services/authentication.service';
import SettingsService from './services/settings.service';
import TaskService from './services/task.service';

const SHARED_PIPES: any[] = [
  FormattedTimePipe,
  QueuedOnlyPipe
];

const SHARED_PROVIDERS: any[] = [
  AuthenticationService,
  SettingsService,
  TaskService
];

export {
  Queueable,
  Task,

  FormattedTimePipe,
  QueuedOnlyPipe,
  SHARED_PIPES,

  AuthenticationService,
  SettingsService,
  TaskService,
  SHARED_PROVIDERS
};
```

Как следует из определения фасада, новая служба является частью группы с псевдонимом `SHARED_PROVIDERS`. Теперь она будет доступна инструментам внедрения зависимостей всего приложения, поскольку этот символ объявлен в массиве поставщиков зависимостей корневого компонента.

Вернемся к классу службы. Здесь импортируется декоратор `Injectable`. Как вы уже знаете, он необходим для автоматического создания и внедрения в компоненты экземпляра класса `AuthService` как синглтона (на случай, если в дальнейшем классу потребуются зависимости). Кроме того, мы импортировали класс `EventEmitter`, который будет рассмотрен ниже в этом разделе.

В теле класса `AuthenticationService` определены пустой конструктор и три метода без реализации (один из них статический). Хотя имена очевидно описывают назначение методов, последний из них требует дополнительных пояснений. Метод `isAuthorized()` информирует о наличии у пользователя права доступа к защищенным стра-

ницам. Он объявлен статическим потому, что будет использоваться там, куда не дотягивается механизм внедрения зависимостей Angular и невозможно автоматическое внедрение зависимостей.

Итак, согласно первому требованию, нам нужно реализовать общедоступный метод, обслуживающий вход пользователей. Начнем с него. Вернемся в модуль `AuthenticationService` и добавим в метод `login` следующую реализацию:

APP/SHARED/SERVICES/AUTHENTICATION.SERVICE.TS

```
login({username, password}): Promise<boolean> {
  return new Promise(resolve => {
  let validCredentials: boolean = false;

  // @ПРИМЕЧАНИЕ: в реальном приложении
  // это должно выполняться удаленной службой:
  if (username === 'john.doe@mail.com' &&
      password === 'letmein') {
        validCredentials = true;
        window.sessionStorage.setItem('token', 'eyJhbGciOi');
    }
    resolve(validCredentials);
  });
}
```

Как следует из комментария, данные для проверки допустимости не передаются удаленной веб-службе, хотя это непременно следует сделать. Нелишним будет еще раз вспомнить, о чем говорилось в предыдущих главах: никогда не выполняйте проверку пользователя приведенным здесь способом. А теперь перейдем к рассмотрению реализации метода. Первое, что привлекает внимание, – тип возвращаемого значения. Этот метод возвращает объект `Promise`, и тому есть веская причина. Обычно при реализации асинхронного HTTP-подключения к удаленной службе ей отправляются учетные данные пользователя и ожидают от нее ответа. Следовательно, используется асинхронный интерфейс в виде возвращаемого объекта `Promise`, который разрешается в логическое значение, информирующее о возможности доступа к системе с указанными учетными данными. С другой стороны, в сигнатуре метода отсутствует аннотация типа аргумента, но раскладка объекта указывает, что метод принимает любой объект, имеющий свойства `username` и `password`. И последнее, но важное замечание: сразу после проверки подлинности в хранилище сеансов браузера пользователя сохраняется маркер со случайным значением. В настоящее

время это самый распространенный способ аутентификации и сохранения сеанса, с той лишь разницей, что маркер извлекается браузером из тела ответа сервера, а затем включается в заголовки всех запросов к серверу.

Описание серверной реализации выходит за рамки этой книги и поэтому не будет здесь рассматриваться. Более подробную информацию по этой теме можно найти в других книгах издательства Packt.

После знакомства с обработкой входа пользователя перейдем к методу `logout()`, который выполняет обратную операцию:

APP/SHARED/SERVICES/AUTHENTICATION.SERVICE.TS

```
logout(): Promise<boolean> {
  return new Promise(resolve => {
    window.sessionStorage.removeItem('token');
    resolve(true);
  });
}
```

Согласно третьему требованию, мы должны реализовать свойство или метод, сообщающий о прохождении пользователем проверки подлинности, чтобы позволить ему перейти на защищенные страницы. Учитывая, что права пользователя определяются наличием или отсутствием маркера безопасности в хранилище браузера, логика метода весьма проста:

```
static isAuthorized(): boolean {
  return !!window.sessionStorage.getItem('token');
}
```

Мы обсудим этот метод и разберемся, почему он объявлен статическим, ниже. А пока займемся последним требованием к службе: добавим наблюдаемый объект, позволяющий элементам пользовательского интерфейса и другим клиентам в приложении подписаться на событие изменения состояния пользователя. Сначала определим общедоступное свойство `EventEmitter`, которое будет использоваться для отправки уведомлений при входе и выходе пользователя из системы, чтобы другие классы и компоненты могли подписаться и реагировать на эти события. Очевидно, что в методы `login` и `logout` требуется добавить отправку соответствующих уведомлений в зависимости от выполненных ими действий и текущего состояния пользователя.

Ниже приводится окончательный вариант службы аутентификации, которая может подключаться как зависимость:

```
import { Injectable, EventEmitter } from '@angular/core';

@Injectable()
export default class AuthenticationService {
  userIsloggedIn: EventEmitter<boolean>;

  constructor() {
    this.userIsloggedIn = new EventEmitter();
  }

  login({ username, password }): Promise<boolean> {
    return new Promise(resolve => {
      let validCredentials: boolean = false;

      // @ПРИМЕЧАНИЕ: в реальном приложении эта проверка
      // должна выполняться удаленной службой:
      if (username === 'john.doe@mail.com' &&
        password === 'letmein') {
        validCredentials = true;
        window.sessionStorage.setItem('token', 'eyJhbGciOi');
      }

      this.userIsloggedIn.emit(validCredentials);
      resolve(validCredentials);
    });
  }

  logout(): Promise<boolean> {
    return new Promise(resolve => {
      window.sessionStorage.removeItem('token');
      this.userIsloggedIn.emit(false);
      resolve(true);
    });
  }

  static isAuthorized(): boolean {
    return !!window.sessionStorage.getItem('token');
  }
}
```

Импортирование новой службы в других компонентах

Наличие поставщика делает службу аутентификации доступной через инструмент внедрения зависимостей приложения, и мы можем начинать подключать ее к другим компонентам. Наиболее логичным местом ее использования является точка запуска приложения. Итак, откройте модуль `LoginComponent` и импортируйте новый тип

AuthenticationService, чтобы получить возможность внедрить его экземпляр в компонент:

APP/LOGIN/LOGIN.COMPONENT.TS

```
import { Component } from '@angular/core';
import {
  FormBuilder,
  ControlGroup,
  Validators,
  Control } from '@angular/common';
import { Router } from '@angular/router-deprecated';
import { AuthenticationService } from '../shared/shared';
...
```

Здесь же внесем изменения в конструктор, добавив новый аргумент, аннотированный типом AuthenticationService, чтобы сообщить механизму внедрения зависимостей, какой модуль использовать:

```
constructor(
  formBuilder: FormBuilder,
  private router: Router,
  private authService: AuthService) {
  // Остальная часть конструктора не изменилась
  ...
}
```

Теперь можно удалить прежнюю логику из метода authenticate(), заменив ее следующей:

```
authenticate() {
  let credentials: any = this.loginForm.value;
  this.notValidCredentials = !this.loginForm.valid &&
                             this.loginForm.dirty;

  this.authenticationService.login(credentials).then(success => {
    if (success) {
      this.router.navigateByUrl('/');
    } else {
      this.notValidCredentials = true;
    }
  });
}
```

Блокирование несанкционированного доступа

Пришло время закрыть незарегистрированным пользователям доступ к защищенному содержимому. В нашем случае в защите от несанкцио-

нированных запросов нуждается лишь форма редактирования заданий. В предыдущей главе мы видели, как разрешить или запретить создание экземпляра компонента с помощью маршрутизатора `Router`.

Эти знания делают защиту компонента редактирования заданий от нежелательных визитов довольно простой задачей. Откройте файл `TaskEditorComponent`, импортируйте нового поставщика `AuthenticationService` и осуществите проверку пользователя, связав вызов статического метода `isAuthorized()` с декоратором `CanActivate`:

APP/TASKS/TASK-EDITOR.COMPONENT.TS

```
...
// Другие операторы импорта остаются прежними
import {
  Task,
  TaskService,
  AuthenticationService } from '../shared/shared';

@Component({
  selector: 'pomodoro-tasks-editor',
  directives: [ROUTER_DIRECTIVES],
  providers: [Title],
  templateUrl: 'app/tasks/task-editor.component.html',
  styles: [`
        .ng-valid { border-color: #3c763d; }
        .ng-invalid { border-color: #a94442; }
        .ng-untouched { border-color: #999999; }
    `]
})
@CanActivate(AuthService.isAuthorized)
export default class TaskEditorComponent implements OnActivate,
CanDeactivate, OnDeactivate {
  // Реализация класса не изменилась
  ...
}
```

Все готово! Теперь если незарегистрированный пользователь попытается получить доступ к защищенному компоненту редактора заданий, у него ничего не выйдет! Внимательно рассмотрев декоратор `@CanActivate()`, можно понять, почему метод `isAuthorized()` класса `AuthenticationService` определен как статический. Причина в том, что внедрять синглтоны в качестве зависимостей можно только в компоненты, но не в декораторы. Впрочем, это не совсем верно (для привязки нужного синглтона можно использовать метод `provide()`, правда,

при этом придется написать сложный и запутанный код), однако приведенная реализация проста и при той же эффективности выглядит аккуратнее и проще.



В идеале следовало бы внедрить поставщиков `AuthenticationService` и `Router` в реализацию `CanActivate`, а затем перенаправлять пользователя на страницу входа, если он еще не вошел в систему.
К сожалению, на момент написания книги все еще отсутствовала официальная поддержка внедрения зависимостей в контекст события `CanActivate` маршрутизатора. Однако это одна из дополнительных возможностей, которые войдут в окончательную версию Angular 2. Вполне вероятно, что в окончательной версии декоратор `@CanActivate` будет заменен аналогичным методом экземпляра родительского компонента маршрутизации. Следите за официальной документацией.

Реакция пользовательского интерфейса на состояние аутентификации пользователя

Здорово, что несанкционированные пользователи не могут получить доступа к форме редактирования заданий. Однако наличие бесполезной ссылки в главной панели инструментов не радует, поэтому воспользуемся возможностью наблюдения за `AuthenticationService` для переключения пользовательского интерфейса при любом изменении состояния аутентификации пользователя.

Прямо сейчас займемся ссылкой `Login` в навигационной панели, указывающей на страницу с формой входа. Нам нужно скрыть ссылку публикации задания `Publish Task` и отображать ее только после входа пользователя в систему, независимо от того, где и как была выполнена процедура входа. С другой стороны, мы должны вывести ссылку `Logout`, чтобы зарегистрировавшийся пользователь смог закрыть сеанс в конфиденциальном порядке. Эта ссылка должна быть доступна лишь вошедшим пользователям. Отклонение попыток доступа к защищенному компоненту посредством ввода прямого URL-адреса в адресной строке браузера не является проблемой, поскольку `@CanActivate` сам отклонит запросы нежелательных пользователей.

Теперь, после перечисления требований, займемся их воплощением. Откройте файл корневого компонента и внесите изменения в его реализацию (сейчас она пуста). Нам потребуются зависимости `AuthenticationService` и `Router`, поэтому импортируем их в начале файла:

APP/APP.COMPONENT.TS

```
import { Component } from '@angular/core';
import {
  SHARED_PROVIDERS,
  AuthenticationService } from './shared/shared';
import { HTTP_PROVIDERS } from '@angular/http';
import {
  ROUTER_PROVIDERS,
  RouteConfig,
  ROUTER_DIRECTIVES,
  Router } from '@angular/router-deprecated';
// Остальные операторы импорта остаются прежними
...
```

Импортировав требуемые имена, можно двигаться дальше и создать реализацию класса `AppComponent`:

APP/APP.COMPONENT.TS

```
...
export default class AppComponent {
  userIsLoggedIn: boolean;

  constructor(
    private authenticationService: AuthenticationService,
    private router: Router) {
      authenticationService.userIsloggedIn.subscribe(isLoggedIn => {

        this.userIsLoggedIn = isLoggedIn;
      });
    }

    logout($event): void {
    $event.preventDefault();

    this.authenticationService.logout().then(success => {
      if (success) {
        this.router.navigateByUrl('/');
      }
    });
  }
}
```

Мы объявили логическое поле `userIsLoggedIn`, отражающее изменения значения наблюдаемого свойства `userIsloggedIn` объекта `AuthenticationService`. Мы также внедрили тип `Router` и создали новый метод `logout()`, закрывающий сеанс пользователя и отправляющий его на главную страницу.

Теперь завершим доработку пользовательского интерфейса приложения, изменив шаблон корневого компонента, чтобы заставить ссылки реагировать на состояние пользователя:

APP/APP.COMPONENT.HTML

```
<nav class="navbar navbar-default navbar-static-top">
  <div class="container">
    <div class="navbar-header">
  <strong class="navbar-brand">My Pomodoro App</strong>
</div>
<ul class="nav navbar-nav navbar-right">
  <li><a [routerLink]="['TasksComponent']">Tasks</a></li>
  <li><a [routerLink]="['TimerComponent']">Timer</a></li>
  <li *ngIf="userIsLoggedIn">
    <a [routerLink]="['TaskEditorComponent']">
      Publish Task
    </a>
  </li>
  <li *ngIf="!userIsLoggedIn">
    <a [routerLink]="['LoginComponent']">Login</a>
  </li>
  <li *ngIf="userIsLoggedIn">
      <a href="#" (click)="logout($event)">Logout</a>
    </li>
  </ul>
  </div>
</nav>
<router-outlet></router-outlet>
```

Давайте попробуем! Перезагрузите приложение, проверьте ссылки на панели навигации, перейдите на страницу входа, выполните вход с правильными учетными данными и проверьте еще раз навигационную панель... Волшебство!

Улучшение управления доступом

На настоящий момент в приложении имеется все необходимое. Но по мере его роста будет увеличиваться число областей, нуждающихся в защите, что значительно затруднит переключение видимости все большего и большего количества ссылок, а также регулирование доступа к компонентам с помощью добавления декоратора `@CanActivate` к каждому из них.

Очевидно, что это отрицательно влияет на масштабируемость, и намного лучше было бы иметь более универсальное решение. К со-

жалению, на момент написания книги фреймворк Angular 2 все еще не обеспечивал решения этой проблемы. С другой стороны, согласно современным концепциям организации взаимодействия с пользователем, следует предоставлять ему как можно больше возможностей и отправлять его на страницу входа, только если это действительно необходимо.

В этом последнем разделе мы рассмотрим обходное решение данной проблемы, основанное на следующем критерии: защищенные области должны охватываться дочерними маршрутами, отправляющими пользователя на страницу входа при обнаружении несанкционированного доступа, причем такие параметры, как путь к странице входа, должны полностью настраиваться из решения.

Разработка собственной директивы безопасности RouterOutlet

Свое обходное решение мы реализуем на основе собственной директивы `RouterOutlet`, расширяющей класс `RouterOutlet`, встроенный в Angular 2, немного изменив поведение по умолчанию базовой директивы в отношении принятия решения об активации затребованного компонента. Решения будут приниматься, исходя из текущего состояния пользователя.

Для этого создадим новую директиву в общем контексте, которая будет использоваться во всем приложении, и экспортируем ее в общем фасаде.

APP/SHARED/DIRECTIVES/ROUTER-OUTLET.DIRECTIVE.TS

```
import {
  Directive,
  ViewContainerRef,
  DynamicComponentLoader,
  Attribute,
  Input } from '@angular/core';
import {
  Router,
  RouterOutlet,
  ComponentInstruction } from '@angular/router-deprecated';
import { AuthenticationService } from '../shared';

@Directive({
  selector: 'pomodoro-router-outlet'
})
export default class RouterOutletDirective extends RouterOutlet {
```

```
parentRouter: Router;
@Input() protectedPath: string;
@Input() loginUrl: string;

constructor(
  _viewContainerRef: ViewContainerRef,
  _loader: DynamicComponentLoader,
  _parentRouter: Router,
  @Attribute('name') nameAttr: string) {

    super(_viewContainerRef, _loader, _parentRouter, nameAttr);
    this.parentRouter = _parentRouter;
  }

activate(nextInstruction: ComponentInstruction): Promise<any> {
  let requiresAuthentication =
  this.protectedPath === nextInstruction.urlPath;

  if (requiresAuthentication &&
  !AuthenticationService.isAuthorized()) {
    this.parentRouter.navigateByUrl(this.loginUrl);
  }

return super.activate(nextInstruction);
  }
}
```

Мы определили новую директиву, расширяющую стандартную директиву `RouterOutlet`, использовавшуюся в приложении раньше. Эта директива должна быть доступна для применения в прочих контекстах приложения:

APP/SHARED/SHARED.TS

```
import Queueable from './interfaces/queueable';
import Task from './interfaces/task';

import FormattedTimePipe from './pipes/formatted-time.pipe';
import QueuedOnlyPipe from './pipes/queued-only.pipe';

import AuthenticationService from './services/authentication.service';
import SettingsService from './services/settings.service';
import TaskService from './services/task.service';

import RouterOutletDirective from './directives/router-outlet.
directive';

const SHARED_PIPES: any[] = [
  FormattedTimePipe,
```

```
  QueuedOnlyPipe
];

const SHARED_PROVIDERS: any[] = [
  AuthenticationService,
  SettingsService,
  TaskService
];

const SHARED_DIRECTIVES: any[] = [
  RouterOutletDirective
];

export {
  Queueable,
  Task,

  FormattedTimePipe,
  QueuedOnlyPipe,
  SHARED_PIPES,

  AuthenticationService,
  SettingsService,
  TaskService,
  SHARED_PROVIDERS,

  RouterOutletDirective,
  SHARED_DIRECTIVES
};
```

Вернитесь в файл нашей директивы и обратите внимание, что она наследует конструктор директивы `RouterOutlet`. Поэтому мы импортировали все имена, необходимые для объявления зависимостей в конструкторе, и вызвали конструктор суперкласса с помощью метода `super()`:

APP/SHARED/DIRECTIVES/ROUTER-OUTLET.DIRECTIVE.TS

```
constructor(
    elementRef: ElementRef,
  _loader: DynamicComponentLoader,
  _parentRouter: Router,
  @Attribute('name') nameAttr: string) {

  super(_elementRef, _loader, _parentRouter, nameAttr);
  this.parentRouter = _parentRouter;
}
```

В теле конструктора не только внедряются зависимости, но и присваивается свойству `parentRouter` экземпляр класса `Router`, который

понадобится позднее. Все эти изменения сделаны в основном ради переопределения встроенного метода `activate()`, в который мы добавили проверку пользователя (именно поэтому мы импортировали `AuthenticationService` в начале сценария), чтобы закрыть доступ к компоненту, входящему в защищенную область. В данном случае, если статический метод `isAuthorized()` службы `AuthenticationService` выявит, что пользователь не авторизован, этот пользователь будет отправлен на страницу входа. В любом случае, метод вернет результат вызова метода `activate` суперкласса – объект `Promise`:

```
activate(nextInstruction: ComponentInstruction): Promise<any> {
  let requiresAuthentication =
      this.protectedPath === nextInstruction.urlPath;

  if (requiresAuthentication &&
      !AuthenticationService.isAuthorized()) {
    this.parentRouter.navigateByUrl(this.loginUrl);
  }

  return super.activate(nextInstruction);
}
```

Откуда взялись параметры `protectedPath` и `loginUrl`? Как упоминалось в начале раздела, директива экспортирует два входных параметра, то есть ее экземпляры будут выглядеть следующим образом:

```
<pomodoro-router-outlet protectedPath="edit" loginUrl="/login">
</pomodoro-router-outlet>
```

Теперь откройте файл корневого компонента и замените текущий экземпляр директивы `RouterOutlet` в шаблоне следующими строками, поместив их непосредственно после инструкции импорта символа `SHARED_DIRECTIVES` в свойстве `directives` декоратора компонента:

APP/APP.COMPONENT.TS

```
import { Component } from 'angular2/core';
import {
  SHARED_PROVIDERS,
  AuthenticationService,
  SHARED_DIRECTIVES } from './shared/shared';
  ...

@Component({
  selector: 'pomodoro-app',
  directives: [ROUTER_DIRECTIVES, SHARED_DIRECTIVES],
  ...
})
```

APP/APP.COMPONENT.HTML

```
<nav class="navbar navbar-default navbar-static-top">
  <div class="container">
    <div class="navbar-header">
      <strong class="navbar-brand">My Pomodoro App</strong>
    </div>
    <ul class="nav navbar-nav navbar-right">
      <li><a [routerLink]="['TasksComponent']">Tasks</a></li>
      <li><a [routerLink]="['TimerComponent']">Timer</a></li>
      <li *ngIf="userIsLoggedIn">
        <a [routerLink]="['TaskEditorComponent']">
          Publish Task
        </a>
      </li>
      <li *ngIf="!userIsLoggedIn">
        <a [routerLink]="['LoginComponent']">Login</a>
      </li>
      <li *ngIf="userIsLoggedIn">
        <a href="#" (click)="logout($event)">Logout</a>
      </li>
    </ul>
  </div>
</nav>
<pomodoro-router-outlet
  protectedPath="tasks/editor"
  loginUrl="login">
</pomodoro-router-outlet>
```

И наконец, прежде чем опробовать это решение, необходимо сделать еще две вещи: *удалить* (или закомментировать) декоратор `@CanActivate()` в компоненте редактора заданий и настроить обработчик `routerCanDeactivate()` маршрутизатора:

APP/TASKS/TASK-EDITOR.COMPONENT.TS

```
routerCanDeactivate(
  next: ComponentInstruction,
  prev: ComponentInstruction) {
  return !AuthenticationService.isAuthorized() ||
    this.changesSaved ||
    confirm('Are you sure you want to leave?');
}
```

Это позволит без помех деактивировать компонент, если пользователь не вошел в систему. Для чего это сделано? Как правило, перенаправление пользователя будет выглядеть следующим образом:

- пользователь запрашивает защищенный компонент;
- директива маршрутизатора проверяет подлинность пользователя;
- если вход в систему выполнен, директива ничего не делает, и затребованный компонент активируется;
- если нет, компонент все равно будет активирован, но пользователь будет отправлен на страницу входа.

Чтобы проверить это решение в действии, удалите условие из параметров маршрута. Нам нужно показать ссылку, но переадресовать пользователя, если он не вошел в систему:

APP/APP.COMPONENT.HTML

```
...
<ul class="nav navbar-nav navbar-right">
  <li><a [routerLink]="['TasksComponent']">Tasks</a></li>
  <li><a [routerLink]="['TimerComponent']">Timer</a></li>
  <li>
    <a [routerLink]="['TaskEditorComponent']">
      Publish Task
    </a>
  </li>
  <li *ngIf="!userIsLoggedIn">
    <a [routerLink]="['LoginComponent']">Login</a>
  </li>
  <li *ngIf="userIsLoggedIn">
    <a href="#" (click)="logout($event)">Logout</a>
  </li>
</ul>
...
```



У этого решения имеется один важный недостаток: перед переадресацией пользователя на страницу входа (где запрашиваются учетные данные) защищенный компонент выводится на экран. Это, безусловно, не очень хорошо, так как есть шанс, что защищенное содержимое появится на экране, если создание экземпляра безопасного компонента займет больше времени, чем ожидается, или метод `canDeactivate()`, при определенных условиях, запретит дальнейшее движение.
Однако обычно переход выполняется настолько быстро, что пользователь даже не успевает заметить созданного компонента. Но, как бы то ни было, используйте это решение с осторожностью и тщательно следите за реализацией функций `CanDeactivate`.

Итоги

Несомненно, эта глава была длинной и довольно сложной. Наиболее важный вывод этой главы заключается в том, что в Angular 2 существует множество взаимоисключающих вариантов проектирования и реализации форм. Одни из них основываются на декларативном подходе, другие – на императивном. Когда следует использовать тот или иной подход? Как уже упоминалось выше, все зависит от того, где и каким образом требуется получить доступ или решить проблему доступа к свойствам состояния и проверки допустимости объектов `Control` и `ControlGroup`.

Кроме того, в этой главе были рассмотрены проблемы аутентификации пользователя и представлены различные способы защиты закрытых областей сайта. Следите за последними усовершенствованиями Angular 2, поскольку вполне вероятно, что применение обходных приемов со временем потеряет свою актуальность.

А теперь готовьтесь к последнему этапу путешествия по фреймворку Angular 2, в ходе которого мы рассмотрим поддержку анимации. А завершится это путешествие освоением технологий модульного тестирования, которому будет посвящена последняя глава книги.

Глава 9

Анимация компонентов в Angular 2

В настоящее время анимация является одной из основ проектирования восприятия сайта пользователями. Из простого украшательства пользовательского интерфейса она превратилась в важный аспект визуального представления информации. Анимация пошла по пути ненавязчивой передачи сообщений, став создающим минимум затрат, но мощным средством информирования пользователя об основных процессах и событиях, сопровождающих его взаимодействие с приложением. На настоящий момент, когда анимация широко распространена и целевая аудитория воспринимает ее как должное, необходимо относиться к ней как к важному средству улучшения восприятия приложений. Анимация не привязана к конкретному языку программирования, определенному устройству или среде (настольным, мобильным или веб-приложениям) и хорошо воспринимается пользователями при разумном применении. Другими словами, анимация имеет право на существование, и фреймворк Angular 2 демонстрирует твердую приверженность этому аспекту современной визуальной разработки.

Для всех современных браузеров, поддерживающих новые возможности спецификации CSS3 для обработки анимации, фреймворк Angular 2 предоставляет поддержку императивной реализации анимации сценариями, посредством невероятно простого, но мощного программного интерфейса. В этой главе будет рассмотрено несколько подходов к реализации анимационных эффектов, начиная от исполь-

зования стандартной CSS-разметки для анимации, основанной на классах, до написания модулей сценариев, где фреймворк Angular 2 берет на себя всю заботу о преобразовании модели DOM.

В этой главе рассматриваются:

- анимация с помощью стандартной CSS-разметки;
- использование классов анимации с помощью директивы `ngClass` для улучшения обработки переходов;
- встроенные классы CSS для определения промежуточных стилей в процессе выполнения переходов;
- анимация компонентов с помощью `CssAnimationBuilder`;
- директивы анимации;
- знакомство с ngAnimate 2.0.



Следует отметить, что текущая реализация анимации в Angular 2, имевшаяся на момент написания книги, носит в определенной степени временный характер. В этом смысле вся функциональность, описанная в этой главе, может быть признана устаревшей после выпуска окончательной версии Angular 2. Однако в настоящий момент нет никаких оснований ограничивать использование уже имеющихся модулей поддержки анимации. Это придаст разрабатываемым приложениям функциональную мощь, отвечающую возросшим требованиям пользователей.

Анимация с помощью стандартной CSS-разметки

Появление анимации, основанной на CSS-разметке, стало важной вехой в современном веб-дизайне. До этого для создания анимации в веб-приложениях использовался JavaScript, позволяющий манипулировать элементами DOM посредством сложных и громоздких сценариев, основанных на промежутках времени, задержках и циклах всех видов. К сожалению, такой подход не был ни удобным, ни масштабируемым.

Затем современные браузеры освоили новые возможности, привнесенные CSS-преобразованиями, переходами, ключевыми кадрами и анимационными свойствами. Это стало важным моментом, изменившим правила игры в интерактивном веб-дизайне. Если поддержка этих технологий в таких браузерах, как Microsoft Internet Explorer, все еще далека от совершенства, то прочие браузеры (включая Edge от Microsoft) обеспечивают полную поддержку прикладных программных интерфейсов CSS.



Microsoft Internet Explorer поддерживает эти технологии анимации только начиная с версии 10.

Здесь предполагается, что вам хорошо знакомы принципы работы CSS-анимации, основанной на ключевых кадрах или на переходах, поскольку освещение этих тем, естественно, выходит за рамки данной книги. В качестве резюме можно выделить тот факт, что анимация на основе CSS обычно реализуется с применением одного из перечисленных ниже подходов или их комбинаций:

- **Трансформируемые свойства**, действующие как наблюдатели для всех или только некоторых CSS-свойств, применяются к элементам DOM, затронутым селектором. При изменении любого из этих CSS-свойств элемент DOM не сразу получает новое значение, а происходит постепенный переход в новое состояние.
- **Именованные ключевые кадры** определяют различные этапы с уникальными именами эволюции одного или нескольких CSS-свойств, которые затем переносятся в свойство `animation` заданного селектора, что дает возможность указать дополнительные параметры, такие как задержка, продолжительность или количество итераций анимации.

Как видите, в обоих подходах использование CSS-селектора вместе с параметрами анимации является отправной точкой для определения анимации, и именно это будет здесь реализовано. Мы создадим фантастическую анимацию, имитирующую эффект пульсации растрового изображения, украшающего Pomodoro-таймер.

Поскольку анимация в нашем примере будет основана на ключевых кадрах, начнем с создания обычной CSS-разметки в отдельной таблице стилей. Вся анимация будет опираться на простую интерполяцию, согласно которой заданный объект увеличивается в размерах на 10%, а затем возвращаться в исходное состояние. Эта анимация затем получит имя и будет заключена в CSS-класс `pulse`, выполняющий ее в бесконечном цикле, где каждая итерация занимает 1 секунду.

Все CSS-правила для реализации этой анимации будут располагаться во внешней таблице стилей для компонента таймера в папке `timer`:

APP/TIMER/TIMER-WIDGET.COMPONENT.CSS

```
@keyframes pulse {
  0% {
    transform: scale3d(1, 1, 1);
  }

  50% {
```

```
    transform: scale3d(1.1, 1.1, 1.1);
  }

  100% {
    transform: scale3d(1, 1, 1);
  }
}

.pulse {
  animation: pulse 1s infinite;
}
```

Теперь любой элемент DOM (в шаблоне `TimerWidgetComponent`), аннотированный именем этого класса, будет пульсировать как сердце. Такой визуальный эффект является хорошей иллюстрацией, что элемент выполняет какие-то действия, поэтому применение его к главному растровому изображению значка Pomodoro в виджете Pomodoro-таймера создает ощущение активной работы.

К счастью, имеется удобный способ применения такого эффекта, только когда идет обратный отсчет времени. Используем для этого привязку свойства `isPaused` в шаблоне `TimerWidgetComponent` и с помощью директивы `NgClass` будем присваивать класс `pulse`, только когда компонент не приостановлен, так что откройте файл модуля таймера, добавьте туда ссылку на недавно созданную таблицу стилей и примените директиву, как описано выше:

APP/TIMER/TIMER-WIDGET.COMPONENT.TS

```
...
@Component({
  selector: 'pomodoro-timer-widget',
  styleUrls: ['app/timer/timer-widget.component.css'],
  template: `
    <div class="text-center">
      <img src="/app/shared/assets/img/pomodoro.png"
      [ngClass]="{ pulse: !isPaused }">
      <h3><small>{{ taskName }}</small></h3>
      <h1> {{ minutes }}:{{ seconds | number: '2.0' }} </h1>
      <p>
        <button (click)="togglePause()" class="btn btn-danger">
          {{ buttonLabelKey | i18nSelect: buttonLabelsMap }}
        </button>
      </p>
    </div>`
})
...
```

Все готово! Запустите приложение Pomodoro, щелкните на расположенной сверху ссылке на таймер, чтобы перейти на страницу компонента, и убедитесь в появлении визуального эффекта после начала обратного отчета. Остановите и запустите его, чтобы удостовериться, что эффект применяется, только когда ведется обратный отсчет.

Анимация с помощью классов-событий CSS

Простота применения визуальных эффектов на основе CSS-классов, как было продемонстрировано в предыдущем разделе, достигается главным образом благодаря гибким возможностям фреймворка Angular 2, позволяющим добавлять имена пользовательских классов.

Angular 2 поддерживает также возможность обработки событий анимационных классов, функционально схожую с применяемой в Angular 1.x, но с другой реализацией. Суть механизма заключается в следующем: элементы DOM, управляемые шаблонными директивами (`NgSwitch`, `NgFor` и `NgIf`), можно декорировать классом `ng-animate`. В результате они будут получать дополнительные классы в зависимости от этапа анимации, в контексте жизненного цикла компонента.

Такое определение может выглядеть непонятным и сложным, поэтому рассмотрим все сказанное на реальном примере. Откройте шаблон `TasksComponent` и внесите изменения в тег строки списка заданий, добавив класс `ng-animate`. Затем сделайте то же самое с надписью `queued`, отображаемой, когда задание ставится в очередь:

APP/TASKS/TASKS.COMPONENT.HTML

```
<tr *ngFor="let task of tasks; let i = index" class="ng-animate">
  <th scope="row">{{i}}
    <span *ngIf="task.queued" class="label label-info ng-animate">
      Queued
    </span>
  </th>
  <!-- остальная часть шаблона без изменений -->
</tr>
```

Сохраните и запустите приложение и откройте панель инструментов разработчика в браузере. Ладно, видимо, ничего не происходит. Здесь речь идет об анимации, и основным процессом является именно она, поэтому добавим правила для класса `ng-animate` в таблицу стилей компонента:

APP/TASKS/TASKS.COMPONENT.CSS

```
h3, p {
  text-align: center;
}
.table {
  margin: auto;
  max-width: 860px;
}
.ng-animate {
  transition: all 10s ease-in;
}
```

Это CSS-правило обеспечит плавное применение к элементу DOM любых изменений в стилях, в течение 10 секунд по алгоритму `ease-in`. Запустите код снова и загляните в разметку: как видите, теперь ко всем элементам с классом `ng-animate` присоединено несколько временных классов. Это классы `ng-enter` и `ng-enter-active`, причем первый из них по умолчанию подключается уже при отображении элемента, а второй присоединяется сразу вслед за этим. По истечении 10 секунд, определяющих интервал преобразования, как определено правилом CSS, имена обоих классов исчезнут, оставив класс `ng-animate` в качестве свидетельства своего присутствия.

В сущности, здесь происходит все то же самое, что было реализовано вручную в предыдущем разделе, за исключением того, что привязкой имен классов управляет фреймворк Angular 2. Учитывая это, внесем небольшие изменения в таблицу стилей, чтобы сократить время присутствия этих классов в модели DOM, и настроим свойство `transition` для создания красивого эффекта плавного появления при загрузке. Заменим период преобразования в 10 секунд на более короткий, длительностью в 0,3 секунды (300 миллисекунд), и зададим значения непрозрачности для начального `ng-enter` и конечного `ng-enter-active` классов:

APP/TASKS/TASKS.COMPONENT.CSS

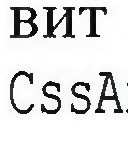
```
...
.ng-animate {
  transition: all 0.3s ease-in;
}
.ng-enter {
  opacity: 0;
}
.ng-enter-active {
  opacity: 1;
}
```

Сохраните и запустите приложение. Теперь список заданий плавно появляется на экране. Это относится и к синей метке, информирующей о постановке задания в очередь.

Поддерживаемые классы-события

Классы CSS, представленные выше, называются в Angular классами-событиями. Управление ими осуществляется уже рассмотренными ранее директивами или событиями жизненного цикла маршрутизатора, они присутствуют в элементах DOM, только пока выполняется анимация. Всего имеются четыре таких класса:

- `ng-enter`: применяется фреймворком Angular к любому элементу DOM с классом `ng-animate` после его присоединения к представлению. Обычно используется для определения оформления компонентов по умолчанию;
- `ng-enter-active`: присоединяется к элементу с началом анимации и автоматически удаляется вместе с именем класса `ng-enter` после ее завершения. Обычно здесь определяется стиль, который нужно применить в конце анимации, ранее сброшенный с помощью класса `ng-enter`;
- `ng-leave`: действует подобно классу `ng-enter`, но добавляется в элемент перед его исключением из представления;
- `ng-leave-active`: действует подобно классу `ng-enter-active`, но добавляется в элемент перед его исключением из представления. Этот класс применяется к элементу и удаляется вместе с `ng-leave` после завершения преобразования, но до того, как элемент будет исключен из дерева DOM.



Помните, что эта технология применима только к элементам DOM, которые обрабатываются классом `DomRenderer` – низкоуровневым классом, отвечающим, в частности, за создание, изменение и удаление узлов и представлений. В данном случае доступными элементами являются только элементы с директивами `NgIf`, `NgFor` или `NgSwitch`.

Анимация компонентов с помощью класса AnimationBuilder

Если вы пожелаете разобраться в закулисной работе классов-событий CSS после привязки класса `ng-animate`, вас наверняка удивит тот факт, что все заботы принимает на себя экземпляр класса `CssAnimationBuilder`, создаваемый с помощью класса `AnimationBuilder`.

Класс `AnimationBuilder` (который является внедряемым и, следовательно, может импортироваться через конструкторы компонентов) является фабрикой, позволяющей создавать более специализированные классы анимации, такие как класс `ssAnimationBuilder`. Этот класс обладает широкими возможностями, позволяющими добавлять и удалять имена CSS-классов, для запуска преобразований или анимации, а также настраивать вручную параметры анимации, такие как стили, длительность или задержка, для отдельных элементов DOM. То есть можно определить обработчики анимации общего назначения, а затем использовать их как адаптеры анимации для любых элементов DOM. В этом случае обработчики анимации, созданные с помощью класса `CssAnimationBuilder`, не зависят от характеристик элементов DOM. Следовательно, один и тот же адаптер анимации может применяться как к одному, так и к нескольким элементам HTML.

Итак, для создания анимационного эффекта программным способом исключительно на JavaScript нам потребуется экземпляр типа `AnimationBuilder`, с помощью которого мы получим экземпляр `CssAnimationBuilder` для создания конкретного (но применимого к любым элементам HTML) анимационного преобразования. И последнее, но важное замечание: нам понадобится также тип для доступа к анимируемым элементам DOM. Выглядит сложно? Все это разъяснит приведение простого примера.

Проверим новые приемы создания новых визуальных эффектов, применив их к Pomodoro-таймеру. Создадим анимационный эффект плавного проявления компонента при каждой активации маршрута `PomodoroTimer`.

Начнем с импорта новых типов в блоке инструкций импорта класса `TimerWidgetComponent`. Мы должны импортировать класс `ElementRef` из барреля `angular/core`, чтобы получить доступ к элементу DOM, который нужно анимировать. Затем следует импортировать класс `AnimationBuilder` для создания экземпляра объекта `CssAnimationBuilder`. Последний также нужно импортировать, чтобы мы могли аннотировать свойства нашего класса:

APP/TIMER/TIMER-WIDGET.COMPONENT.TS

```
import { Component, OnInit, ElementRef } from '@angular/core';
import { RouteParams, CanReuse, OnReuse } from
  '@angular/router-deprecated';
import { SettingsService, TaskService } from '../shared/shared';
```

```
import { AnimationBuilder } from '@angular/platform-browser/src/
  animate/animation_builder';
import { CssAnimationBuilder } from '@angular/platform-browser/src/
  animate/css_animation_builder';
...
```



Обратите внимание, откуда импортируются `AnimationBuilder` и `CssAnimationBuilder`. На момент написания книги баррель `@angular/animate` еще не был включен ни в один из пакетов фреймворка Angular 2, поэтому для импорта символов приходится использовать полный путь. Это может измениться в будущем, поэтому обратитесь к официальной документации Angular 2 при получении ошибки 404 в процессе импорта классов.

Заполучив все необходимые инструменты, можно приступать к созданию основы для анимации. Начнем с создания нового свойства `fadeInAnimationBuilder` в классе контроллера компонента – оно будет хранить экземпляр объекта `CssAnimationBuilder`, определяющий анимационное преобразование. Сначала внедрим зависимости через конструктор класса, добавив модификаторы доступа, чтобы превратить параметры конструктора в свойства класса:

APP/TIMER/TIMER-WIDGET.COMPONENT.TS

```
...
export default class TimerWidgetComponent implements OnInit, CanReuse,
OnReuse {
  minutes: number;
  seconds: number;
  isPaused: boolean;
  buttonLabelKey: string;
  buttonLabelsMap: any;
  taskName: string;
  fadeInAnimationBuilder: CssAnimationBuilder;

  constructor(
    private settingsService: SettingsService,
    private routeParams: RouteParams,
    private taskService: TaskService,
    private animationBuilder: AnimationBuilder,
    private elementRef: ElementRef) {
    this.buttonLabelsMap = settingsService.labelsMap.timer;

    this.fadeInAnimationBuilder = animationBuilder.css();
    this.fadeInAnimationBuilder.setDuration(1000)
      .setDelay(300)
      .setFromStyles({ opacity: 0 })
```

```
        .setToStyles({ opacity: 1 });
  }
  // Остальная часть класса без изменений
}
```

Конструктор вызывает метод `AnimationBuilder.css()`, чтобы создать экземпляр объекта `CssAnimationBuilder`, и тут же присваивает его свойству `fadeInAnimationBuilder`. После этого объект `CssAnimationBuilder` используется для определения анимации: воспроизведение эффекта должно начаться через 300 миллисекунд и продолжаться в течение 1000 миллисекунд (то есть одну секунду), а суть его заключается в плавном изменении любого заданного элемента DOM от полной прозрачности до абсолютной непрозрачности. Следует отметить, что объект `CssAnimationBuilder` поддерживает возможность составления цепочек из вызовов методов, поэтому все настройки можно объединить в удобную последовательность.

Как уже упоминалось в начале раздела, обработчик анимации можно применять к любым элементам. Давайте посмотрим, как применить это преобразование к реальному элементу DOM. Запуск анимации можно выполнить вызовом метода `start()` класса `CssAnimationBuilder`. Этот метод принимает HTML-элемент, к которому должна быть применена подготовленная анимация. Перейдите к обработчику `ngOnInit()` и добавьте в конец следующий блок кода:

APP/TIMER/TIMER-WIDGET.COMPONENT.TS

```
...
ngOnInit(): void {
  this.resetPomodoro();
  setInterval(() => this.tick(), 1000);

  let taskIndex = parseInt(this.routeParams.get('id'));
  if (!isNaN(taskIndex)) {
    this.taskName = this.taskService.taskStore[taskIndex].name;
  }
  this.fadeInAnimationBuilder.start(

    this.elementRef.nativeElement.firstElementChild);
}
...
```

Как уже упоминалось выше, метод `start()` требует передачи HTML-элемента для анимации, что мы и делаем, передав в вызов первый дочерний узел свойства `nativeElement` объекта `elementRef`. Класс `ElementRef`, импортированный в конструкторе, – это ссылка на компо-

нент директивы (компонент PomodoroTimer) в контексте родительского представления или действующего в настоящий момент шаблона. Его свойство nativeElement открывает доступ к фактическому элементу компонента, который обычно является деревом узлов HTML-шаблона. Далее нужно извлечь значение свойства firstElementChild, ссылающееся на корневой узел шаблона, и применить к нему подготовленную анимацию. Важно отметить, что компонент никак не будет реагировать, если анимацию применить непосредственно к нему. Поэтому приходится использовать свойство nativeElement, чтобы получить фактический элемент DOM. Здесь можно использовать другие селекторы DOM, используя методы элементов, такие как document.getElementById() или document.querySelector(), но делать это не рекомендуется, поскольку при этом создается жесткая связь между уровнями контроллера и отображения, что усложнит сопровождение компонентов в будущем.

Сохраните проделанную работу, запустите приложение Pomodoro и перейдите к Pomodoro-таймеру. Вуаля! Теперь таймер плавно проявляется на экране.

CssAnimationBuilder

Мы только что рассмотрели создание независимого обработчика анимации с помощью объекта CssAnimationBuilder и использовали несколько его методов (таких как setDelay, setDuration, setFromStyles и setToStyles). Это лишь часть из обширного множества его методов, включающего также дополнительные методы для создания сложной анимации. Ниже перечислены некоторые из этих методов.

- setDelay(delay: number): как было показано в примере выше, этот метод определяет задержку перед анимацией, переопределяя любые другие задержки, объявленные в CSS-правилах;
- setDuration(duration: number): задает продолжительность анимации и, подобно методу setDelay(), переопределяет любые другие времена, объявленные в CSS-правилах;
- setFromStyles(from: {[key: string]: any}): как было показано в примере выше, этот метод устанавливает начальные значения стилей в виде объекта, содержащего пары ключ/значение. Обратите внимание, что все имена CSS-свойств должны указываться в верблюжьей нотации, то есть все имена CSS-свойств следует преобразовать в верблюжью нотацию. Таким образом, свойства, такие как margin-top или background-color, должны указываться как marginTop или backgroundColor;

- `setToStyles(to: {[key: string]: any})`: определяет конечные значения стилей;
- `setStyles(from: {[key: string]: any}, to: {[key: string]: any})`: синтаксический сахар для прямого доступа к функциональности, предоставляемой методами `setFromStyles` и `setToStyles`, позволяющий определять начальные и конечные состояния стилей анимации;
- `addClass(className: string)`: добавляет класс, остающийся в элементе после завершения анимации. Этот метод особенно полезен для переопределения CSS-свойств элементов DOM, уже управляемых CSS-трансформацией;
- `removeClass(className: string)`: действует подобно предыдущему методу, но удаляет класс из элемента;
- `addAnimationClass(className: string)`: добавляет временный класс, который будет удален в конце анимации. Фреймворк Angular 2 внутренне использует этот метод для обработки CSS-событий, генерируемых привязкой класса `ng-animate` и описанных в начале главы;
- `start(element: HTMLElement)`: запускает анимацию HTML-элемента, указанного в аргументе, и возвращает объект `Animation`. Объект `Animation` имеет очень полезные методы, которые можно использовать для реализации некоторых дополнительных возможностей, в частности для запуска обратных вызовов, после завершения анимации.

Эти методы, способные объединяться в цепочки, позволяют конструировать по-настоящему сложные анимационные эффекты и упрощают их многократное использование во всем приложении.

Отслеживание состояния анимации с помощью класса Animation

Взаимодействие с приложением не прекращается по завершении анимации. Действительно, момент завершения одной анимации может стать началом множества других анимаций и интерактивных событий, происходящих в пользовательском интерфейсе. Поэтому важно иметь возможность обнаружить завершение анимации в приложениях.

К счастью, такую возможность дают класс `Animation` и метод `CssAnimationBuilder.start()`, возвращающий экземпляр этого типа, как показано в следующем примере:

APP/TIMER/TIMER-WIDGET.COMPONENT.TS

```
...
ngOnInit(): void {
  this.resetPomodoro();
  setInterval(() => this.tick(), 1000);

  let taskIndex = parseInt(this.routeParams.get('id'));
  if (!isNaN(taskIndex)) {
    this.taskName = this.taskService.taskStore[taskIndex].name;
  }

  const animation = this.fadeInAnimationBuilder.start(
    this.elementRef.nativeElement.firstElementChild);

  animation.onComplete(() => console.log('Animation completed!'));
}
```

Класс `Animation` дает возможность определить свой обработчик событий `onComplete`, возникающих непосредственно после завершения анимации, запущенной вызовом `CssAnimationBuilder.start()`. Его можно использовать для запуска любых других действий, например аутентификации пользователя (как было показано в предыдущем примере) или следующей анимации.

Фактически всю тяжелую работу по управлению преобразованиями выполняет класс `Animation`. То есть класс `CssAnimationBuilder` – это всего лишь фасад, обеспечивающий дружественный интерфейс для настройки анимации. Но если говорить о выполнении операций в процессе анимации или сразу после ее завершения, в этом может помочь только класс `Animation`.

Тем не менее нам редко придется взаимодействовать с классом `Animation`, иначе как с использованием функций обратного вызова или встроенных методов для обработки классов CSS или замены стилей, после завершения анимации. С другой стороны, этот класс не является внедряемым, поэтому его экземпляры не могут создаваться системой внедрения зависимостей в Angular. По этим причинам мы не будем детально исследовать его программный интерфейс.

Разработка пользовательских директив анимации

Как уже неоднократно упоминалось, одним из преимуществ `CssAnimationBuilder` является возможность его повторного использования. Объединение настроек анимации значительно облегчает их дальней-

шее применение к HTML-элементам. Однако для управления многократным использованием в Angular лучше всего подходят директивы. Так почему бы не объединить вместе их лучшие качества? Как демонстрируется в следующем примере, обертывание анимации директивой становится идеальным решением во многих ситуациях.

В последнем примере в этой главе демонстрируется создание совершенно новой директивы `Highlight`, использующей класс `CssAnimationBuilder` для изменения цвета фона любого заданного элемента DOM на лету, сбрасывая цвет фона в исходное состояние по завершении анимации. Такой эффект *мерцания* довольно широко распространен в современном веб-дизайне и используется для оповещения пользователя о том, что в настоящий момент что-то произошло в некоторой области пользовательского интерфейса.

Создадим файл класса контроллера директивы в подпапке `directives` внутри папки `shared` и поместим в него следующий ниже сценарий. Обратите внимание, что мы по-прежнему следуем соглашению об именовании файлов, а также отметьте, как наша директива отображается в селектор класса CSS:

APP/SHARED/DIRECTIVES/HIGHLIGHT.DIRECTIVE.TS

```
import { Directive, ElementRef, OnInit } from '@angular/core';
import { AnimationBuilder } from '@angular/platform-browser/src/
animate/animation_builder';
import { CssAnimationBuilder } from '@angular/platform-browser/src/
animate/css_animation_builder';

@Directive({
  selector: '.pomodoro-highlight',
  providers: [AnimationBuilder]
})
export default class HighlightDirective {
  cssAnimationBuilder: CssAnimationBuilder;

  constructor(
    private animationBuilder: AnimationBuilder,
    private elementRef: ElementRef) {

    this.cssAnimationBuilder = animationBuilder.css()
      .setDuration(300)
      .setToStyles({ backgroundColor: '#fff5a0' });
  }
  ngOnInit() {
    let animation = this.cssAnimationBuilder.start(
```

```
      this.elementRef.nativeElement
    );

    animation.onComplete(() => {
      animation.applyStyles({ backgroundColor: 'inherit' });
    });
  }
}
```

Код реализации довольно прост. По сути, мы создали директиву, соответствующую селектору класса CSS, чей конструктор запускает CSS-анимацию – смену цвета фона заданным тоном желтого цвета (определенным в виде шестнадцатеричного значения `#fff5a0`) на 300 миллисекунд. Это и есть эффект *мерцания*. Всякий раз, когда компонент с этой директивой отображается в представлении, метод-обработчик `ngOnInit` запускает анимацию, а затем сбрасывает цвет фона элемента DOM в исходное значение. Как видите, для этого используется метод `applyStyles()` класса `Animation`. Этот метод наряду с другими, такими как `addClasses()` или `removeClasses()` (которым требуется массив строк с именами классов для их добавления или удаления, соответственно), позволяет управлять присоединением CSS-правил к анимируемым элементам DOM.

Прежде чем двигаться дальше, необходимо обеспечить доступ к этой новой директиве из остальной части приложения, поэтому экспортируем ее в общем фасадном модуле:

APP/SHARED/SHARED.TS

```
import Queueable from './interfaces/queueable';
import Task from './interfaces/task';

import FormattedTimePipe from './pipes/formatted-time.pipe';
import QueuedOnlyPipe from './pipes/queued-only.pipe';

import AuthenticationService from './services/authentication.service';
import SettingsService from './services/settings.service';
import TaskService from './services/task.service';

import RouterOutletDirective from './directives/router-outlet.directive';
import HighlightDirective from './directives/highlight.directive';
const SHARED_PIPES: any[] = [
  FormattedTimePipe,
  QueuedOnlyPipe
];

const SHARED_PROVIDERS: any[] = [
```

```
  AuthenticationService,
  SettingsService,
  TaskService
];

const SHARED_DIRECTIVES: any[] = [
  RouterOutletDirective,
  HighlightDirective
];

export {
  Queueable,
  Task,

  FormattedTimePipe,
  QueuedOnlyPipe,
  SHARED_PIPES,

  AuthenticationService,
  SettingsService,
  TaskService,
  SHARED_PROVIDERS,

  RouterOutletDirective,
  HighlightDirective,
  SHARED_DIRECTIVES
};
```

Как видите, в обновленном фасаде директива `Highlight` стала частью псевдонима `SHARED_DIRECTIVES`. Как следствие она доступна любому компоненту, включающему этот псевдоним в свое свойство `directives`, в том числе в компоненте `TasksComponent`, чей шаблон будет подвергнут дополнительной настройке.

Откройте шаблон компонента и добавьте в элемент `ngFor` дополнительный класс:

APP/TASKS/TASKS.COMPONENT.HTML

```
<tr *ngFor="let task of tasks; let i = index"
  class="ng-animate highlight">
```

Перезагрузите приложение и наслаждайтесь, наблюдая, как список заданий мигает при загрузке на экран и затем возвращается в обычное состояние. Имейте в виду, что эту функциональность можно применить к любой части дерева DOM приложения, импортировав директиву и добавив имя класса в выбранный элемент DOM. Однако вас, вероятно, интересует, почему для реализации простого эффекта

мерцания потребовалось так много шаблонного кода. Не проще было бы ограничиться CSS-классом? Да, проще.., если вы не предполагаете взаимодействовать с анимацией.

Взаимодействие с директивами шаблона

Справедливости ради следует отметить, что сама по себе директива запуска анимации не имела бы особого смысла, если бы не давала возможности взаимодействовать с анимацией. Более того, было бы неплохо иметь возможность взаимодействовать с ней непосредственно из шаблона. Для этого присвоим директиве экспортируемое имя, чтобы получить к нему доступ из любого элемента с директивой. Помните, как использовалась ссылка в директиве `ngForm` при обработке форм? Здесь будет применяться та же технология. Но сначала внесем изменения в директиву `Highlight`, добавив новое свойство `exportAs` для настройки директивы, имеющее значение `highlight`. Это значение станет именем, по которому можно будет получать доступ к программному интерфейсу директивы за ее пределами. Как это сделать? Немного терпения, сначала скроем метод `ngOnInit`, изменив его имя на `colorize`, в связи с чем удалим класс из первой строки в блоке импорта и реализацию интерфейса из класса:

APP/SHARED/DIRECTIVES/HIGHLIGHT.DIRECTIVE.TS

```
import { Directive, ElementRef } from '@angular/core';
import { AnimationBuilder } from '@angular/platform-browser/src/
animate/animation_builder';
import { CssAnimationBuilder } from '@angular/platform-browser/src/
animate/css_animation_builder';

@Directive({
  selector: '.pomodoro-highlight',
  providers: [AnimationBuilder],
  exportAs: 'pomodoroHighlight'
})
export default class HighlightDirective {
  cssAnimationBuilder: CssAnimationBuilder;

  constructor(
    private animationBuilder: AnimationBuilder,
    private elementRef: ElementRef) {

    this.cssAnimationBuilder = animationBuilder.css()
      .setDuration(300)
      .setToStyles({ backgroundColor: '#fff5a0' });
```

```
  }

  colorize() {
    let animation = this.cssAnimationBuilder.start(
      this.elementRef.nativeElement
    );

    animation.onComplete(() => {
      animation.applyStyles({ backgroundColor: 'inherit' });
    });
  }
}
```

Запустите пример. Теперь таблица не мигает при загрузке, что и требовалось. После внесения описанных выше изменений можно переходить к шаблону. Во-первых, добавим локальную ссылку с именем `row` (или любым другим именем на ваш выбор) в HTML-узел с директивой, указывающую на имя `highlight`, которое, как мы уже знаем, является общедоступным именем директивы. Подобно тому, как с помощью директивы `ngForm` в формах осуществлялся доступ к состоянию, например для проверки допустимости, здесь можно получить доступ к общедоступному программному интерфейсу директивы, включающему метод `colorize()`, созданный из бывшего метода `ngOnInit`. Теперь можно безопасно вызвать этот метод с помощью локальной ссылки:

APP/TASKS/TASKS.COMPONENT.HTML

```
<tr *ngFor="let task of tasks; let i = index"
    class="ng-animate pomodoro-highlight"
    #row="pomodoroHighlight"
    (click)="row.colorize()">
```

Перезагрузите приложение и щелкните на любом задании, включив или исключив его из очереди. Строка, на которой произведен щелчок, мигнет.



В реализации директивы `Highlight` мы жестко зафиксировали CSS-значение. Для большей гибкости и масштабируемости такого подхода следует избегать. Попробуйте в качестве самостоятельного упражнения внести изменения в директиву, применив селектор атрибута, такой как `[pomodoroHighlight]`, значение которого извлекается свойством класса, аннотированного декоратором `@Input`, что позволит настроить пользовательские цвета для мерцания при привязке директивы в шаблонах компонентов.

Взгляд в будущее: ngAnimate 2.0

Большинство рассмотренных в этой главе процедур наследует логику анимации из Angular 1.x, основанную на CSS-классах и анимации ключевыми кадрами, предварительно созданными в таблицах стилей. К сожалению, этот подход нелучшим образом проявляет себя, когда дело доходит до оптимизации производительности, реализации параллельной анимации или продвинутых взаимодействий, подобных предложенным в Material Design[1], и это не полный список его недостатков.

Вот почему команда Angular теперь работает над новым набором модулей, которые должны обеспечить беспрецедентную производительность и абстракцию за счет использования **предметно-ориентированного языка (Domain Specific Language, DSL)** с полноценным прикладным программным интерфейсом. Этот проект примет форму так называемого модуля **ngAnimate 2.0**, который планируется выпустить до конца 2016 года. Новый подход станет программной системой, которая вернется к JavaScript, отделит CSS и обеспечит всеобщее программное управление.

Ниже кратко перечислены главные преимущества ngAnimate 2.0:

- использование фабрик анимации посредством прикладного программного интерфейса с поддержкой параллельной и последовательной анимации;
- полная обработка анимации только на JavaScript, перенос логики CSS-анимации ключевыми кадрами на JavaScript, соединение с помощью универсальных последовательностей и анимационных цепочек;
- значительно меньшие потери производительности;
- анимация в многоуровневой модели DOM и управление звуком через CSS;
- улучшенная интеграция событий и полный контроль над потоком анимации с возможностью быстрой перемотки вперед или назад посредством программной интерполяции;
- полноценная поддержка Material Design и улучшенная обработка событий взаимодействий с пользователем, щелчков мыши и т. д.;
- адаптивная стилизация;
- полноценное тестирование и поддержка анимационных ресурсов в тестовых спецификациях.

---

[1] https://ru.wikipedia.org/wiki/Material_Design. – *Прим. ред.*

Эти и многие другие функции станут доступны с выходом ngAnimate 2.0. На момент выхода книги описание прикладного программного интерфейса еще не опубликовано, что не позволяет его здесь осветить. Следите за новостями, поскольку ngAnimate 2.0 станет важной вехой в обработке анимации на JavaScript.

Итоги

В этой главе мы обсудили несколько методов реализации анимации в Angular 2. Сначала мы рассмотрели способ, основанный на именах классов с трансформациями CSS3, реализованный в виде встроенных директив Angular. Затем были описаны встроенные средства Angular, обеспечивающие взаимодействие с этим видом анимации и облегчающие реализацию собственных трансформаций с применением некоторых соглашений к таблице стилей. Далее речь шла об анимации, управляемой программно с помощью инструментов, имеющихся на настоящий момент в Angular 2. Было продемонстрировано использование обратных вызовов класса `Animation` для обработки событий в течение жизненного цикла анимации.

Последний раздел этой главы был посвящен краткому описанию модуля ngAnimate 2.0, который в ближайшем будущем должен заменить устаревшие технологии.

В следующей и последней главе мы подведем итоги всего, что узнали в предыдущих главах, познакомившись с одной из насущных потребностей, возникающих при разработке современных веб-приложений: модульным тестированием. Angular 2 поддерживает общие отраслевые стандарты и библиотеки, предлагая конкретные инструменты для облегчения модульного тестирования.

Глава 10

Модульное тестирование в Angular 2

В результате напряженной работы в предыдущих главах мы получили действующее приложение, которым можно гордиться. Но как облегчить его дальнейшее обслуживание? Комплексное автоматизированное тестирование превратится в спасательный круг, как только приложение начнет расти, и потребуется смягчить последствия ошибок, вызванных новыми функциями, при наложении их на уже имеющиеся ошибки.

Тестирование (более конкретно: модульное тестирование) должно систематически осуществляться в процессе разработки проекта. Однако детали тестирования модулей Angular 2 будут кратко рассмотрены в этой главе, когда проект уже почти готов.

Здесь описывается порядок реализации инструментов для модульного тестирования классов и компонентов приложения.

В этой главе мы:

- оценим важность тестирования вообще и модульного тестирования в частности;
- рассмотрим особенности модульного тестирования на JavaScript;
- познакомимся с платформой тестирования Jasmine;
- узнаем, как настроить среду модульного тестирования с помощью Jasmine и SystemJS;
- создадим тестовые спецификации для тестирования;
- спроектируем модульные тесты для компонентов с зависимостями и без них;

- протестируем маршруты;
- реализуем тесты для служб, фиктивные зависимости и заглушки;
- научимся перехватывать XHR-запросы и возвращать фиктивные ответы для улучшения управления;
- узнаем, что директивы тестируются как компоненты, только без представлений;
- познакомимся с другими идеями и инструментами, такими как Karma, инструменты оценки охвата кода или тестирование E2E.

Почему тестирование необходимо?

Что такое модульное тестирование? Если вы уже знакомы с идеей модульного тестирования и разработкой, основанной на тестировании, можете пропустить этот раздел. Если нет, давайте считать модульные тесты частью инженерной философии, которая утверждает, что для эффективной и гибкой разработки требуется создать средство автоматизированного тестирования кода прежде, чем он будет написан. Основная идея состоит в том, что каждый фрагмент кода должен сопровождаться тестом, который создается разработчиком этого кода. Сначала создается тест будущего модуля, проверяющий выходные данные и логику поведения. Изначально тест будет оканчиваться неудачей, поскольку модуль еще не реализован. Следовательно, наша работа будет заключаться в том, чтобы создать модуль, способный пройти свой тест.

Модульное тестирование является весьма спорной темой. Хотя и существует всеобщее согласие в вопросе полезности разработки, основанной на тестах, для обеспечения нужного качества кода и сокращения времени на его обслуживание, но далеко не все применяют модульное тестирование в повседневной практике. Чем это вызвано? Создание тестов при разработке может иногда восприниматься как ненужная обуза, особенно когда тест превышает в объеме тестируемый им код.

Но аргументы «за» более весомы, чем аргументы «против»:

- создание тестов способствует улучшению архитектуры кода. Код должен соответствовать требованиям теста и не может содержать обходных путей. В этом смысле, если попытки тестирования существующего фрагмента кода в определенный момент стабильно оканчиваются неудачей, велик шанс, что

тестируемый фрагмент кода плохо проработан и имеет запутанный интерфейс, требующий переосмысления. С другой стороны, разработка тестируемых модулей помогает раннему выявлению побочного влияния кода на другие модули;

- тестирование вносимых изменений является единственным способом, помогающим избежать появления ошибок на более поздних этапах. Разработка подразумевает развитие с течением времени, и любое изменение сопряжено с риском внесения ошибок, которые проявятся в других областях приложения. Модульные тесты помогают выявить ошибки на ранней стадии, либо при добавлении новых функций, либо при изменении существующих;
- создание тестов является хорошим способом документирования программного интерфейса и его возможностей. И ценность такого документирования значительно возрастает, когда за доработку кода берется кто-то, кому код незнаком.

Это лишь некоторые из аргументов, но бесчисленное их количество можно найти на сайтах в Интернете, посвященных пользе тестирования. Если приведенные аргументы не кажутся вам вескими, попробуйте поискать сами. В противном случае будем двигаться дальше и рассмотрим общую форму тестов.

Средства модульного тестирования в Angular 2

Существует много различных способов тестирования фрагментов кода, но основное наше внимание будет уделено модульному тестированию в Angular 2. Первое, что потребуется, – это фреймворк для тестирования, предоставляющий служебные функции для создания *наборов тестов*, содержащих один или несколько *тестов*.

```
describe('The submit component', () => { // Набор тестов
  it('should be rendered disabled by default', () => {
    // Тест
    // ... Здесь находится реализация теста
  });

});
```

Каждый *тест* проверяет конкретную функциональность, описанную в определении набора тестов, и объявляет одно или несколько

ожиданий в своем теле. Каждое *ожидание* содержит значение, которое называется *ожидаемым* значением и сравнивается с *фактическим* значением посредством функции *сопоставления*, проверяющей совпадение ожидаемого и фактического значений. В результате образуется то, что называется *утверждением*, и фреймворк тестирования считает тест успешным или неуспешным, в зависимости от результата такого утверждения. Например:

```
describe('The submit component', () => { // Набор тестов

  it('should be rendered disabled by default', () => {
    // Тест

    // Утверждение, основанное на экземпляре компонента
    expect(submitComponent.disabled).toBe(true);

  });

});
```

В предыдущем примере `submitComponentInstance.disabled` вернет *фактическое* значение, которое должно соответствовать *ожидаемому*, объявленному в функции сопоставления `toBe()`.

Внедрение зависимостей в модульных тестах

Довольно часто перед выполнением тестов необходимо произвести ряд операций: создать экземпляр класса компонента или службы, извлечь зависимости, объявить фиктивный аргумент и т. д. Наиболее распространенной операцией при модульном тестировании в Angular 2 является переопределение поставщиков по умолчанию с помощью функции `beforeEachProviders()`. Эта функция должна выполняться *до* любой другой и имеет следующую форму:

```
describe('The submit component', () => {

  beforeEachProviders(() => [
    TestComponentBuilder,
    MyCustomService,
    provide(Router, { useClass: RootRouter })
  ]);

  it('should be rendered disabled by default', () => {
    expect(submitComponent.disabled).toBe(true);
  });

});
```

В этом примере, до выполнения любого из тестов, настраивается список поставщиков, необходимых для внедрения зависимостей. Как видите, можно просто объявить имена классов или даже переопределить зависимости, привязав имя к классу, значению или фабричной функции на выбор с помощью функции `provide()`. На самом деле подмена зависимостей фиктивными классами с фиктивными данными или функциональностью является распространенной практикой при тестировании.

А как насчет создания экземпляров объектов, требуемых для тестирования, или подготовки испытательных данных, необходимых для любого теста? Функция `beforeEach()` является естественным местом для извлечения зависимостей, ранее объявленных в функции `beforeEachProviders()`, создания экземпляров проверяемых классов или выполнения любых других операций, необходимых для подготовки объектов или данных, требующихся тестам. Она должна вызываться *после* функции `beforeEachProviders()`, но *перед* тестами. Для этого используется функция `inject()`.

В следующем фрагменте подробно разбирается пример, реализующий все необходимые действия. Обратите внимание на комментарии, описывающие назначение каждого блока кода:

```
describe('The submit component', () => {

  // Объявим зависимости для управления поставщиком
  beforeEachProviders(() => [
    TestComponentBuilder
  ]);

  // Переменная для компоновщика теста
  let testComponentBuilder: TestComponentBuilder;

  // Перед каждым тестом из инструмента внедрения зависимостей
  // извлекается экземпляр TestComponentBuilder
  beforeEach(inject([TestComponentBuilder],
                    (tcb: TestComponentBuilder) => {
    testComponentBuilder = tcb;
  })
);

// Аргумент функции done указывает на асинхронность теста
it('should be rendered disabled by default', done => {

  // Компоновщик асинхронно возвращает объект,
  // обертывающий экземпляр класса SubmitComponent
  testComponentBuilder
```

```
.createAsync(SubmitComponent)
.then(testComponent => {

  // Извлечение экземпляра компонента из обертки
  const submitComponent = testComponent.componentInstance;

  // Проверка соответствия с помощью функции сопоставления
  expect(submitComponent.disabled).toBe(true);
  // в заключение вызывается функция done(), асинхронно
    // возвращающая тест
    done();
  });
});
```

Пусть вас не беспокоит класс `TestComponentBuilder`, мы еще вернемся к нему. Обратите особое внимание, как извлекается экземпляр класса `TestComponentBuilder` с помощью функции `inject()`, которая возвращает требуемый экземпляр в аргументе функции обратного вызова, который связывается с переменной во внешней области для последующего использования в тесте. Намного выразительнее извлечение экземпляров объектов выглядит с извлечением инструмента внедрения зависимостей и последующим использованием его вспомогательного метода `get()`:

```
let myCustomService; // Пользовательская служба

beforeEach(inject([Injector], (injector: Injector) => {
    myCustomService = injector.get(MyCustomService);
  })
);
```

Вы можете выбрать любой синтаксис, который посчитаете более удобным. Остальная часть примера в основном связана с общими операциями, необходимыми при тестировании компонентов: создание обертки для тестируемого компонента, извлечение экземпляра компонента из обертки, оценка экземпляра компонента и, наконец, разрешение асинхронного *теста* с помощью команды `done()`. Все это подробно будет рассмотрено при описании тестирования компонентов далее в этой главе.

Существует еще несколько инструментов модульных тестов (*макеты*, *шпионы* и т. п.), и самые распространенные из них будут рассмотрены ниже.

Остались нерешенными еще два важных вопроса: как заставить тесты выполняться и как узнать об их успехе или неудаче? Первая

проблема решается привлечением среды тестирования JavaScript-кода, в качестве которой мы будем использовать Jasmine (http://jasmine.github.io), которой, похоже, отдает предпочтение команда Angular. Что касается второй проблемы: нам понадобится *средство выполнения тестов*, и такое средство, основанное на HTML, входит в состав Jasmine, поэтому рассмотрим настройку этой среды.

Настройка среды тестирования

Прежде всего нужно загрузить среду Jasmine и интегрировать ее в проект. Для этого в окне терминала перейдите в папку проекта и выполните следующую команду `npm`, которая установит пакет `jasmine-core` (включая конкретные версии зависимостей, что определяет флаг `--save-exact`, заменяющий семантику версий по умолчанию):

```
$ npm install jasmine-core --save-dev --save-exact
```

После установки пакета убедитесь, что проект включает типы глобальных объектов и функций сопоставления Jasmine, позволяя компилятору распознавать типы и строить тесты:

```
$ typings install jasmine --save --ambient
```

Имея все необходимые библиотеки, посмотрим с помощью средства выполнения тестов, как запустить тестирование и вывести результаты.

Реализация средства выполнения тестов

Тесты будут выполняться в браузере, поэтому необходимо указать основных поставщиков для тестирования с учетом того, что они разные для различных платформ браузеров. Создайте папку `test` в корневой папке проекта и поместите в нее следующий файл:

TEST/SETUP.TS

```
import { resetBaseTestProviders, setBaseTestProviders } from '@
angular/core/testing';
import { BROWSER_APP_DYNAMIC_PROVIDERS } from "@angular/platform-
browser-dynamic";
import {
  TEST_BROWSER_STATIC_PLATFORM_PROVIDERS,
  ADDITIONAL_TEST_BROWSER_PROVIDERS
} from '@angular/platform-browser/testing';
resetBaseTestProviders();
```

```
setBaseTestProviders(
  TEST_BROWSER_STATIC_PLATFORM_PROVIDERS,
  [
    BROWSER_APP_DYNAMIC_PROVIDERS,
    ADDITIONAL_TEST_BROWSER_PROVIDERS
  ]
);
```

В дальнейшем этот файл будет импортирован в код реализации тестирования, и сейчас нет никакой необходимости вникать в его детали. Просто отметим, что эти поставщики обеспечивают доступ к адаптерам модели DOM, необходимым для выполнения определенных операций с теневой моделью DOM.

Перед тем как двигаться дальше, разберемся с используемым в этой книге соглашением для Angular 2, касающимся именования файлов и размещения тестов. Существует общее правило, требующее хранить файлы тестов там же, где находятся тестируемые модули. Для большей ясности имена файлов тестов будут соответствовать именам тестируемых модулей с добавлением суффикса .spec. Так будет легче найти соответствующие модулю тесты, выделить не имеющие тестов модули и просто быстрее ознакомиться с кодовой базой. Помните, что хороший тест является ценной частью документации.

Применим это соглашение на практике, создав в корневой папке проекта временный тест для проверки средства выполнения тестов:

TEST.SPEC.TS

```
describe('Our test runner', () => {
  it('is alive!', () => {
    expect(true).toBe(true);
  });
});
```

А теперь создадим файл средства выполнения тестов в корневой папке проекта. Средство выполнения очень похоже на главную HTML-страницу, касательно добавляемых в его начало нужных ресурсов сценариев, но вместе с тем к нему следует подключить сценарии Jasmine и пакет тестирования Angular 2. В приведенной ниже реализации объявляются массив со строковыми указателями на местоположение файла настроек тестирования браузера и только что созданный тест, без указания пути к скомпилированным файлам TypeScript:

SPEC-RUNNER.HTML

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="content-type"
      content="text/html;charset=utf-8">
    <title>Pomodoro App Unit Tests</title>
    <link rel="stylesheet"
      href="node_modules/jasmine-core/lib/jasmine-core/jasmine.css">
    <script src="node_modules/jasmine-core/lib/jasmine-core/jasmine.js"></
script>
    <script src="node_modules/jasmine-core/lib/jasmine-core/jasmine-html.
js"></script>
    <script src="node_modules/jasmine-core/lib/jasmine-core/boot.js">
    </script>

    <script src="node_modules/es6-shim/es6-shim.min.js"></script>

    <script src="node_modules/zone.js/dist/zone.js"></script>
    <script src="node_modules/reflect-metadata/Reflect.js"></script>
    <script src="node_modules/systemjs/dist/system.js"></script>
    <script src="node_modules/rxjs/bundles/Rx.js"></script>

    <script src="systemjs.config.js"></script>
  </head>
  <body>
    <script>
      // Здесь указывается значение параметра 'outDir' компилятора typescript
      var outDir = 'built';

      // В следующем массиве размещаются файл импорта настроек и
      // массив коллекции тестов, расширения файлов опущены
      var specCollection = [
        'test/setup',
        'test.spec' // Вновь созданный тест
      ];

      // Асинхронная загрузка всех тестов из папки сборки
      // и их одновременное выполнение
      Promise.all(specCollection.map(specPath => {
        return System.import(`${outDir}/${specPath}`);
      }))
      .then(window.onload)
      .catch(console.error.bind(console));
    </script>
  </body>
</html>
```

Просмотрите список импортируемых сценариев и обратите внимание на последний блок. Как уже упоминалось, мы определили путь к скомпилированным файлам (в соответствии с настройками в файле `tsconfig.json`) и массив строковых путей к основным настройкам тестирования в браузере и тестам, причем в данный момент у нас имеется один-единственный тест. Реализация `System.config` очень проста и похожа на ту, которая была создана для запуска проекта через `index.html`. Последний блок немного сложнее и требует дополнительных пояснений. По существу, в нем создается массив команд `System.import` посредством отображения массива путей к настройкам и тестам в другой массив с добавлением к этим путям папки `outDir`. Каждый вызов метода `System.import()` возвращает объект `Promise`, поэтому окончательный массив команд можно выполнить одной командой `Promise.all()` перед активацией оконного события `onload`. Именно этого события ожидает Jasmine для вывода отчета об удачном или неудачном прохождении тестов.

Пришло время посмотреть, как все это действует! Сохраните изменения, запустите компилятор для сборки файла настроек и недавно созданного теста, а затем, запустив локальный веб-сервер, перейдите на URL-адрес файла средства выполнения тестов в адресной строке браузера. Полученный в результате веб-отчет приведен на рис. 10.1.



Рис. 10.1

Настройка команд NPM

Модульное тестирование является многократно повторяемым процессом, поэтому имеет смысл несколько его облегчить. Для этого создадим несколько обертывающих команд, очищающих содержимое папки для сборки, выполняющих повторную компиляцию проекта и запускающих веб-сервер со ссылкой на средство выполнения тестов. В действительности в файл `package.json` можно поместить команду `test`, которая будет вызывать сценарии `pretest` и `posttest`. Вос-

пользуемся этой информацией и внесем изменения в блок сценариев в файле `package.json`, включив в него команды, выполняющие упомянутые выше операции:

PACKAGE.JSON

```
...
"scripts": {
  "postinstall": "npm run typings install",
  "tsc": "tsc",
  "tsc:w": "tsc -w",
  "lite": "lite-server",
  "prestart": "tsc",
  "start": "concurrently \"npm run tsc:w\" \"npm run lite\" ",
  "typings": "typings",
  "pretest": "rm -rf ./built && npm run tsc",
  "test": "npm run lite --open=/spec-runner.html"
},
...
```

Теперь для запуска всего пакета тестов приложения достаточно выбрать в консоли корневую папку проекта и выполнить команду `npm test`. Имейте в виду, что компилятор не выполняется в режиме просмотра, поэтому во время сеансов разработки предпочтительнее выполнять команду `npm start` и загружать средство тестирования в отдельном окне браузера.

Настройка завершена! Следующие страницы будут посвящены различным примерам тестов для компонентов, директив, фильтров, служб и маршрутов из проекта приложения Pomodoro. В Angular 2 не существует универсальной модели тестирования, и для проверки субъектов разного вида используются разные подходы. Далее мы займемся поиском повторяющихся моделей тестирования, а отличающиеся части реализации будут описываться отдельно, поэтому мы не будем углубляться в детали в конкретных примерах. В конечном счете суть TDD состоит в том, чтобы объяснить работу кода через тесты.

Пользовательские функции сопоставления Angular 2

Какие проверки производятся функциями сопоставления в Angular 2? Их очень много. Кроме наиболее распространенных, таких как `toBe()` или `toEqual()`, можно использовать любые функции сопоставления,

встроенные в Jasmine. Полный перечень доступных функций можно найти на официальном сайте Jasmine. Кроме того, фреймворк Angular 2 реализует набор своих функций сопоставления для выполнения операций, распространенных при модульном тестировании в Angular 2:

- `toBePromise()`: ожидает получить в аргументе объект `Promise`;
- `toBeAnInstanceOf(expected: any)`: ожидает, что фактическое значение аргумента `expected` является экземпляром класса;
- `toHaveText(expected: any)`: ожидает, что элемент содержит текст, указанный в аргументе `expected`;
- `toHaveCssClass(expected: any)`: ожидает, что элемент содержит CSS-класс, указанный в аргументе `expected`;
- `toHaveCssStyle(expected: any)`: ожидает, что элемент включает стиль, указанный в аргументе;
- `toImplement(expected: any)`: ожидает, что класс реализует указанный интерфейс;
- `toContainError(expected: any)`: ожидает исключения с указанным текстом ошибки;
- `toThrowErrorWith(expectedMessage: any)`: ожидает функцию, которая возбудит ошибку с указанным текстом;
- `toMatchPattern(expectedMessage: any)`: ожидает, что строка совпадет с указанным регулярным выражением.

Все перечисленные функции сопоставления возвращают логическое значение. Иногда требуется проверить утверждение, обратное реализуемому функцией. В этом случае просто нужно добавить модификатор `.not` перед сопоставлением:

```
expect(true).not.toBe(false);
```

Тестирование фильтров

Фильтры, по сути, являются классами, реализующими интерфейс `PipeTransform` и, соответственно, имеющими метод `transform()`, который обычно является синхронным. В этом смысле фильтры являются идеальными кандидатами, с которых следует начинать освоение модульного тестирования в Angular 2. Начнем с тестирования фильтра `FormattedTimePipe`, создав тест в папке с файлом модуля. Код теста приводится ниже:

APP/SHARED/PIPES/FORMATTED-TIME.PIPE.SPEC.TS

```
import FormattedTimePipe from './formatted-time.pipe';
import {
  describe,
  expect,
  it,
  beforeEach} from '@angular/core/testing';

describe('shared:FormattedTimePipe', () => {
  let formattedTimePipe: FormattedTimePipe;

  beforeEach(() => formattedTimePipe = new FormattedTimePipe());

  // Тесты с утверждениями
  it('should expose a transform() method', () => {
    expect(typeof formattedTimePipe.transform).toEqual('function');
  });

  it('should transform 50 into "0h:50m"', () => {
    expect(formattedTimePipe.transform(50)).toEqual('0h:50m');
  });

  it('should transform 75 into "1h:15m"', () => {
    expect(formattedTimePipe.transform(75)).toEqual('1h:15m');
  });

  it('should transform 100 into "1h:40m"', () => {
    expect(formattedTimePipe.transform(100)).toEqual('1h:40m');
  });

});
```

Здесь импортируется имя фильтра, чтобы создать его экземпляр и присвоить переменной перед запуском каждого из тестов, которые проверяют тип этой переменной или передают разные значения методу `transform` для последующей сверки с ожидаемыми значениями.

А что насчет цикла тестирования, упоминавшегося выше? В нормальной ситуации нужно было бы начать с добавления теста при проектировании интерфейса фильтра. До создания реализации тест должен был завершаться неудачей. При определении утверждений следует также добавить утверждения, проверяющие, как фильтр справляется с некорректными входными данными или неожиданными ситуациями. В процессе доработки и изменения кода должны меняться и тесты.



Следует отметить, что импорт глобальных функций Jasmine был выполнен из Angular 2, а не из самого Jasmine. Причина этого заключается в том, что Angular 2 переопределяет встроенные функции Jasmine, хотя их функциональность и интерфейс остаются неизменными.

Сохраните файл и добавьте вновь созданный тест в массив `specCollection` (точно так же можно без всякого риска удалять ссылки на не используемые больше тесты):

SPEC-RUNNER.HTML

```
...
var specCollection = [
  'test/setup',
  'app/shared/pipes/formatted-time.pipe.spec'
];
...
```

В следующих разделах будет рассмотрено создание различных тестов.



После создания каждого нового теста необходимо добавить путь (без расширения файла) в массив `specCollection`, в файле `spec-runner.html`. *Не забывайте этого делать*, поскольку далее об этом я не буду напоминать. Если забыть включить ссылку, тест не будет выполнен, а сам тест не будет включен в отчет средства выполнения.

Тестирование компонентов

В отличие от фильтров, тестирование компонентов реализуется сложнее, особенно если это проделывается впервые. При этом возникает масса вопросов, например: как тестировать компонент, если его необходимо куда-то загружать? Для этой цели в Angular 2, а конкретнее в его пакете инструментов тестирования, имеется класс `TestComponentBuilder`, с помощью которого можно создавать компоненты любых типов, завернутые в специальный объект, предоставляющий доступ к экземпляру компонента и его скомпилированному HTML-представлению. Любой экземпляр `TestComponentBuilder` предоставляет следующие свойства и методы:

- `debugElement`: объект `DebugElement`, связанный с корневым элементом компонента;
- `ComponentInstance`: возвращает экземпляр корневого компонента с возможностью доступа ко всем его свойствам и методам;

- `NativeElement`: возвращает корневой элемент компонента;
- `DetectChanges()`: активирует циклическое обнаружение изменений компонента. Этот метод вызывается для проверки влияния на состояние компонента изменений в его свойствах или вызовов его методов;
- `destroy()`: вызывает уничтожение компонента.

С учетом всех этих возможностей создадим первый тест компонента. Идеальным кандидатом на эту роль является компонент `TaskIconsComponent`:

APP/TASKS/TASK-ICONS.COMPONENT.SPEC.TS

```
import TaskIconsComponent from './task-icons.component';
import {
  describe,
  expect,
  it,
  inject,
  beforeEach,
  beforeEachProviders } from '@angular/core/testing';
import { TestComponentBuilder } from '@angular/compiler/testing';

describe('tasks:TaskIconsComponent', () => {
  let testComponentBuilder: TestComponentBuilder;

  // Сначала настроим внедрение зависимостей, чтобы получить
  // доступ к компоненту и компоновщику
    beforeEachProviders(() => [TestComponentBuilder]);

    // Повторное создание компоновщика компонента перед каждым тестом
    beforeEach(
      inject([TestComponentBuilder],
      (_testComponentBuilder: TestComponentBuilder) => {
        testComponentBuilder = _testComponentBuilder;
      }
    ));

// Тесты с утверждениями
it('renders 1 image for each pomodoro session required', done => {
  // Создание тестируемого компонента TaskIconsComponent во время
  // выполнения
  testComponentBuilder
  .createAsync(TaskIconsComponent)
  .then(componentFixture => {

  // Извлечение экземпляров компонента и отображение DOM
```

```
    let taskIconsComponent = componentFixture.componentInstance;
    let nativeElement = componentFixture.nativeElement;

    // Установка тестового значения в свойстве @Input и вызов
    // обнаружения изменений
    taskIconsComponent.task = { pomodorosRequired: 3 };
    componentFixture.detectChanges();

    // Эти утверждения оценивают вывод компонента в модель DOM
      expect(nativeElement.querySelectorAll('img').length).toBe(3);

      // По завершении уничтожаем компонент и асинхронно возвращаем
      // результат теста
      componentFixture.destroy();
      done();

    })
    .catch(e => done.fail(e));
  });

});
```

Обратите внимание на блок операторов импорта в начале файла. Кроме импорта базовых функций тестирования, были импортированы символы, относящиеся к механизму внедрения зависимостей, и функции из пакета тестирования, в частности методы настройки `beforeEach` и `beforeEachProviders`. Первым мы вызвали метод `beforeEachProviders`, которому передали лямбда-функцию, возвращающую массив необходимых поставщиков. Функция `beforeEach` использует инструмент внедрения зависимостей для извлечения экземпляра объекта класса `TestComponentBuilder`, ранее объявленного как поставщик, а затем присваивает его переменной `testComponentBuilder`. Внутри теста переменная объекта используется для вызова асинхронного метода `createAsync()`, возвращающего через объект *Promise* обертку выбранного компонента (переданного в аргументе). Как уже упоминалось в начале раздела, из обертки можно получить фактический экземпляр объекта `TaskIconsComponent` и его корневой элемент.

Затем настраиваются свойства компонента. При изменении значения любого входного свойства или вызове метода, способного изменить состояние компонента, необходимо вызвать метод обертки `detectChanges()` для обнаружения изменений, чтобы отразить эти изменения в состоянии `nativeElement`. Это позволит проверить утверждения и с помощью функции сопоставления сравнить количество узлов DOM с ожидаемым. В заключение мы уничтожаем компонент и возвращаем результат вызовом асинхронной функции теста.

Как правило, набор включает несколько тестов, например по одному на каждую рассматриваемую функцию. Поэтому добавим в набор `describe()` еще один оператор после имеющегося:

APP/TASKS/TASK-ICONS.COMPONENT.SPEC.TS (продолжение)

```
...
it('should render each image with the proper width', done => {
  testComponentBuilder.createAsync(TaskIconsComponent)
  .then(componentFixture => {
    let taskIconsComponent = componentFixture.componentInstance;
    let nativeElement = componentFixture.nativeElement;
    let actualWidth;

    taskIconsComponent.task = { pomodorosRequired: 2 };
    taskIconsComponent.size = 60;
    componentFixture.detectChanges();

    actualWidth = nativeElement
      .querySelector('img')
      .getAttribute('width');
    expect(actualWidth).toBe('60');

    done();
  })
  .catch(e => done.fail(e));
});
...
```

Можно добавить столько тестов, сколько потребуется, чтобы охватить все возможные ситуации.



Отладка самих тестов

При создании все большего и большего числа тестов увеличивается вероятность ошибок в самих тестах, что ведет к общей недостоверности отчета о тестировании. Отслеживание возникающих при этом проблем может быть достаточно сложным делом, поэтому в таких случаях желательно выполнять наборы тестов или отдельные тесты изолированно или временно отключать неработоспособные тесты. Для этого можно использовать перечисленные ниже версии функций `describe()` и `it()`:

- `fdescribe()`: запустит только тесты, входящие в указанную группу. Также может использоваться как `ddescribe()`;
- `fit()`: выполнит только указанный тест, игнорируя все остальные. Также может использоваться как `iit()`;
- `xdescribe()`: исключит указанный набор тестов;
- `xit()`: исключит указанный тест.

Тестирование компонентов с зависимостями

В предыдущем разделе мы реализовали первый модульный тест для проверки простого компонента без зависимостей. Однако компоненты и другие модули Angular 2 обычно имеют внедряемые зависимости. При создании модульного теста необходимо учесть это обстоятельство. Для примера рассмотрим компонент `TimerWidgetComponent`. Этот крошечный компонент имеет множество зависимостей, внедряемых через его конструктор:

APP/TIMER/TIMER-WIDGET.COMPONENT.TS

```
...
constructor(
  private settingsService: SettingsService,
  private routeParams: RouteParams,
  private taskService: TaskService,
  private animationBuilder: AnimationBuilder,
  private elementRef: ElementRef) { ...
```

То есть, чтобы создать экземпляр компонента в обертке, возвращаемой фабрикой `TestComponentBuilder`, нужно объявить поставщиков этих зависимостей в инструменте внедрения зависимостей и внедрить их каким-то образом. Кроме того, компонент имеет некоторые важные особенности, проявляющиеся при его выполнении: он чувствителен к параметрам URL-адреса, и одна из его зависимостей (`TaskService`) выполняет базовые XHR-операции через `Http`-модуль. Их необходимо перехватить и должным образом эмулировать.

Кажется, что реализовать все это довольно сложно, но на самом деле для разработки этого теста требуется лишь собрать воедино уже известные процедуры. Рассмотрим код, обращая внимание на комментарии:

APP/TIMER/TIMER-WIDGET.COMPONENT.TEST.TS

```
import TimerWidgetComponent from './timer-widget.component';
import { provide } from '@angular/core';
import { RouteParams } from '@angular/router-deprecated';
import { SettingsService, TaskService } from '../shared/shared';
import {
  describe,
  expect,
  it,
  inject,
  beforeEach,
```

```
  beforeEachProviders,
  setBaseTestProviders } from '@angular/core/testing';
import { TestComponentBuilder } from '@angular/compiler/testing';
import { Http, BaseRequestOptions } from '@angular/http';
import { MockBackend } from '@angular/http/testing';
import 'rxjs/add/operator/map';

describe('timer:TimerWidgetComponent', () => {
  let testComponentBuilder: TestComponentBuilder;
  let componentFixture: any;

  // Сначала настроим внедрение зависимостей в компонент и компоновщик
  // Примечание: поставщики анимации не нужны
  beforeEachProviders(() => [
    TestComponentBuilder,
    SettingsService,
    TaskService,

    // Создать экземпляр RouteParams с передачей параметров путем внедрения
    provide(RouteParams, {useValue: new RouteParams({id: null})}),

    // Внедренный поставщик Http будет заменен позднее в TaskService
    MockBackend,
    BaseRequestOptions,
    provide(Http, { useFactory:
      (backend:MockBackend, options:BaseRequestOptions) => {
        return new Http(backend, options);
      },
      deps: [MockBackend, BaseRequestOptions]
    }),
    TimerWidgetComponent
  ]);

  // Экземпляр компоновщика компонента создается заново перед каждым тестом
  beforeEach(inject([TestComponentBuilder],
    (_testComponentBuilder: TestComponentBuilder) => {
      testComponentBuilder = _testComponentBuilder;
    }
  ));
});
```

Обратите внимание, что в наборе нет ни одного тестового утверждения. Мы еще вернемся к ним, а пока разберем имеющийся код. Реализация набора тестов содержит уже знакомый код внедрения поставщиков. Сначала вызовом метода `beforeEachProviders()` объявляются все поставщики, необходимые для внедрения. Как уже упоминалось, компонент `TimerWidgetComponent` зависит от объекта `RouteParams`, по-

зволяющего извлечь значение параметра id строки запроса, если таковой имеется. Очевидно, что одного лишь объявления поставщика недостаточно; инструмент внедрения зависимостей должен вернуть экземпляр с параметром, подготовленным для тестирования:

```
provide(RouteParams, {useValue: new RouteParams({id: null})}),
```

Большее внимание требуется уделить пониманию порядка выполнения HTTP-запросов, осуществляемых TaskService. Конструктор по умолчанию модуля Http требует наличия объекта подключения, реализующего интерфейс ConnectionBackend. Для нужд тестирования мы должны предоставить инструмент внедрения зависимостей с рабочей версией Http и, следовательно, использовать MockBackend, который реализует требуемый интерфейс. Далее в этой главе будет показано, как использовать этот класс для перехвата XHR-запросов и возврата ответов, заранее подготовленных для тестов.

```
BaseRequestOptions,
provide(Http, { useFactory:
  (backend:MockBackend, options:BaseRequestOptions) => {
    return new Http(backend, options);
  },
  deps: [MockBackend, BaseRequestOptions]
}),
```

Теперь, когда все поставщики доступны инструменту внедрения зависимостей, их можно использовать для создания экземпляра нового объекта фабрики TestComponentBuilder перед выполнением тестов.

```
beforeEach(inject([TestComponentBuilder],
  (_testComponentBuilder: TestComponentBuilder) => {
    testComponentBuilder = _testComponentBuilder;
  }
));
```

После подготовки окружения тестирования вставим первый тест. Добавьте следующий фрагмент сразу после блока beforeEach(...) внутри набора describe(...):

```
it('should initialise with the pomodoro counter at 24:59', done => {
  // Во время выполнения создается окружение для тестируемого компонента
  testComponentBuilder
  .createAsync(TimerWidgetComponent)
  .then(componentFixture => {

    // Извлечь экземпляр компонента и отображаемое дерево DOM
```

```
let timerWidgetComponent = componentFixture.componentInstance;
let nativeElement = componentFixture.nativeElement;

// Вызвать обработчика OnInit и активировать обнаружение изменений
timerWidgetComponent.ngOnInit();
componentFixture.detectChanges();

// Проверить утверждения о свойствах компонента
expect(timerWidgetComponent.isPaused).toBeTruthy();
    expect(timerWidgetComponent.minutes).toEqual(24);
    expect(timerWidgetComponent.seconds).toEqual(59);

    componentFixture.destroy();
    done();
    // Асинхронный возврат текста
  })
    .catch(e => done.fail(e));
});
```

Как видите, вновь созданный тест выполняется путем создания экземпляра компонента и его обертки с окружением. Вызов метода `ngOnInit()` компонента принудительно инициализирует его, как при отображении в представлении. Затем вызывается метод обертки `detectChanges()`, запускающий циклическое обнаружение изменений в экземпляре компонента, происходящих в процессе выполнения `ngOnInit()`.

Следующие тесты, которые можно добавить в тело функции `describe()` сразу после предыдущего теста, закрепляют эти идеи:

```
it('should initialise displaying the default labels', done => {
  testComponentBuilder
  .createAsync(TimerWidgetComponent)
  .then(componentFixture => {
    componentFixture.componentInstance.ngOnInit();
    componentFixture.detectChanges();

    expect(componentFixture.componentInstance.buttonLabelKey)
      .toEqual('start');
    expect(componentFixture.nativeElement
      .querySelector('button')
      .innerHTML.trim())
      .toEqual('Start Timer');

    componentFixture.destroy();
    done();
  })
  .catch(e => done.fail(e));
});
```

Переопределение зависимостей компонентов для улучшения тестирования

В предыдущих примерах было показано, как объявлять и внедрять поставщиков, необходимых для тестирования. Кроме того, было рассмотрено использование функции `provide()` для получения экземпляра любого заданного поставщика, уже заполненного нужными значениями. В этом смысле функция `provide()` используется не просто для замены зависимостей после внедрения поставщиков, но и для определения отдельного поставщика конкретного типа.

Но можно ли переопределить поставщиков на уровне теста? Да, и это очень полезно для подстановки фиктивных зависимостей в некоторых тестах. Следующий тест продолжит тестирование компонента таймера. Но здесь мы заменим поставщика `TaskService` литералом объекта с фиктивными данными. Кроме того, мы заменим объект `RouteParams`, внедряемый по умолчанию, другим его экземпляром, содержащим конкретное значение в параметре `id`.

Добавьте следующий тест сразу после двух предыдущих внутри функции `describe(...)`:

```
it('should initialise displaying a specific task', done => {
  // Подмена поставщика TaskService фиктивными данными
  let mockTaskService = {
    taskStore: [{
        name: 'Task A'
      }, {
        name: 'Task B'
      }, {
        name: 'Task C'
      }
    ]
  };

  testComponentBuilder
  .overrideProviders(TimerWidgetComponent, [
    provide(RouteParams, { useValue: new RouteParams({ id: '1' }) }),
    provide(TaskService, { useValue: mockTaskService })
  ])
  .createAsync(TimerWidgetComponent)
  .then(componentFixture => {
    componentFixture.componentInstance.ngOnInit();
    componentFixture.detectChanges();

    expect(componentFixture.componentInstance.taskName)
      .toEqual('Task B');
```

```
    expect(componentFixture.nativeElement.querySelector('small'))
      .toHaveText('Task B');

    componentFixture.destroy();
    done();
  })
  .catch(e => done.fail(e));
});
```

Код не требует подробных пояснений, но все же проанализируем следующий его блок:

```
testComponentBuilder.overrideProviders(TimerWidgetComponent, [
  provide(RouteParams, { useValue: new RouteParams({ id: '1' }) }),
  provide(TaskService, { useValue: mockTaskService })
])
```

По сути, метод `overrideProviders()` фабрики `TestComponentBuilder` ожидает получить в первом аргументе тип компонента, поставщика которого требуется переопределить, а во втором аргументе – массив поставщиков. В этом массиве можно перечислить любые типы, переопределяемые или заменяемые с помощью функции `provide()`.

Функция `overrideProviders()` анализирует текущее значение свойства `providers` декоратора компонента, чьи поставщики переопределяются. Если декоратор не имеет этого свойства (например, когда все зависимости наследуются от корневого инструмента внедрения), фреймворк Angular вызывает исключение. Поэтому для нужд данного примера включите пустое свойство `providers` в декоратор конфигурации `TimerWidgetComponent`:

APP/TIMER/TIMER-WIDGET.COMPONENT.TS

```
@Component({
  selector: 'pomodoro-timer-widget',
  styleUrls: ['app/timer/timer-widget.component.css'],
  providers: [],
  template: ` ... `
})
```

Возможно, в дальнейшем эта проблема будет решена командой Angular 2, но пока следует поступать именно так.



Как переопределить директивы или шаблон тестируемого компонента?

Другие элементы тестируемого компонента можно переопределить с помощью методов `overrideTemplate()`, `overrideView()` (позволяет переопределить литералы сущностей, такие как стили)

и `overrideDirectives()`. Их сигнатуры следуют тому же соглашению, согласно которому в первом аргументе передается тип компонента, а во втором – замена оригинального значения.
Более подробную информацию можно найти в официальной документации прикладного программного интерфейса.

Тестирование маршрутов

Маршруты, подобно компонентам, играют важную роль во взаимодействиях приложения с пользователями. Поэтому тестирование маршрутов имеет первостепенное значение для безотказной работы приложения. Попытка объявить поставщика `Router` вызовет исключение, поэтому необходимо каким-то образом уведомить инструмент внедрения зависимостей, что этот тип следует использовать в качестве корневого маршрутизатора и корневого компонента, подключая все зависимости этих двух классов. Кроме того, нужно подключить фиктивную службу `Location` в виде более специализированной службы `SpyLocation`. Следующий пример поможет прояснить все это (пока не обращайте внимания на импорт компонента `LoginComponent`, он будет использован в следующем разделе):

APP/APP.COMPONENT.SPEC.TS

```
import AppComponent from './app.component';
import { LoginComponent } from './login/login';
import { provide } from '@angular/core';
import {
  describe,
  expect,
  it,
  inject,
  beforeEach,
  beforeEachProviders } from '@angular/core/testing';
import {
  Router,
  RouteRegistry,
  ROUTER_PRIMARY_COMPONENT } from '@angular/router-deprecated';
import { Location } from '@angular/common';
import { SpyLocation } from '@angular/common/testing';
import { RootRouter } from '@angular/router-deprecated';

describe('AppComponent', () => {
  let location: Location, router: Router;

  // Переопределяются поставщики Router и Location и их зависимости
  // для создания тестового маршрутизатора, активирующего действия
```

```
// маршрутизации и отслеживание местоположения
beforeEachProviders(() => [
  RouteRegistry,
  provide(Location, { useClass: SpyLocation }),
  provide(Router, { useClass: RootRouter }),
  provide(ROUTER_PRIMARY_COMPONENT, { useValue: AppComponent })
]);

// Создание экземпляров Router и Location перед каждым тестом
beforeEach(inject([Router, Location], (_router, _location) => {
  router = _router;
  location = _location;
}));

// Тесты с утверждениями
it('can navigate to the main tasks component', done => {
  // Переход к компоненту и проверка получившегося URL-адреса
  router.navigate(['TasksComponent'])
    .then(() => {
      expect(location.path()).toBe('/tasks');
      done();
    })
    .catch(e => done.fail(e));
  });
});
```

Тестирование маршрутов с помощью URL-адреса

В предыдущем примере мы протестировали переход по именованному маршруту и проверили состояние полученного в результате URL-адреса маршрутизатора. Но иногда нужно проверить, действительно ли был загружен необходимый компонент. В следующем примере используется противоположный подход: выполняется переход по URL-адресу и проверяется тип полученного компонента. Помните, что мы импортировали тип `LoginComponent`? Теперь используем его для тестирования.

Добавьте следующий тест в файл `app/app.component.spec.ts`:

```
...
it('should be able to navigate to the login component', done => {
  // Переход по URL-адресу и проверка типа полученного компонента
  router.navigateByUrl('/login').then(() => {
    expect(router.currentInstruction.component.componentType)
      .toBe(LoginComponent);
    done();
  }).catch(e => done.fail(e));
});
```

Тестирование переадресации

Как проверить работу переадресации? Не волнуйтесь, для этого потребуется лишь выполнить переход по URL-адресу с переадресацией, а затем проверить вид текущей инструкции маршрутизации или получившийся в результате путь.

Добавьте следующий тест в файл app/app.component.spec.ts:

```
it('should redirect "/" requests to the tasks component', done => {
  // Переход по URL-адресу и проверка состояния URL-адреса
  router.navigateByUrl('/').then(() => {
    expect(location.path()).toBe('/tasks');
    done();
  }).catch(e => done.fail(e));
});
```

Тестирование служб

Службы также являются одним из субъектов тестирования. Одна из уникальных особенностей служб в Angular заключается в том, что они могут не иметь никакого отношения к Angular 2. Если службе необходимо воспользоваться механизмом внедрения зависимостей Angular 2 для доступа к модулям Angular, таким как HTTP, применяется подход, характерный для обычных JavaScript-классов, не зависящих от фреймворка.

Тестирование служб без зависимостей во многом подобно тестированию фильтров, когда достаточно создать экземпляр класса для каждого теста и проверить значения его свойств и функциональность методов.

Рассмотрим пример тестирования уже имеющейся простой службы:

APP/SHARED/SERVICES/SETTINGS.SERVICE.SPEC.TS

```
import SettingsService from './settings.service';
import {
  describe,
  expect,
  it,
  inject,
  beforeEach,
  beforeEachProviders } from '@angular/core/testing';

describe('shared:SettingsService', () => {
  let settingsService: SettingsService;
```

```
beforeEach(() => {
  settingsService = new SettingsService();
});

it('should provide the duration for each pomodoro', () => {
  expect(settingsService.timerMinutes).toBeDefined();
  expect(settingsService.timerMinutes).toBeGreaterThan(0);
  expect(settingsService.timerMinutes).not.toBeNaN();
});

  it('should provide plural mappings for tasks', () => {
    const tasksPluralMappings = settingsService.pluralsMap['tasks'];
    const actualProperties = Object.keys(tasksPluralMappings).sort()
    const expectedProperties = ['=0', '=1', 'other'].sort();

    expect(tasksPluralMappings).toBeDefined();
    expect(actualProperties).toEqual(expectedProperties);
  });
});
```

Здесь проверяется соответствие свойства `timerMinutes` и содержимого `tasksPluralMappings` ожидаемым значениям. Можно (и даже нужно) добавить еще несколько тестов, но пока ограничимся достигнутым и сосредоточимся на деталях. Здесь можно было бы воспользоваться механизмом тестирования в Angular 2 и создать экземпляр объекта `SettingService`:

```
beforeEachProviders(() => [SettingsService]);

beforeEach(inject([SettingsService],
  (_settingsService: SettingsService) => {
    settingsService = _settingsService;
  }
));
```

Этот синтаксис напоминает приведенный в предыдущих разделах. В конечном итоге вы можете применять подход, лучше соответствующий вашему стилю программирования. Очевидно, что последний способ требует меньше изменений при доработке класса службы, но зависимости в этом случае должны внедряться исключительно с помощью функции `beforeEachProviders()`.

Тестирование асинхронных служб

Предыдущий пример демонстрирует тестирование самой простой службы. Но обычно пользовательские службы полагаются на зависимости для поддержки своих функций. Кроме того, эти функции не-

редко используют асинхронные методы для подключения к третьим службам. Независимо от интерфейсов, предоставляемых этими асинхронными методами (обратные вызовы, события, объекты `Promise` или наблюдение), тестирование подобных служб реализуется достаточно просто, как показывает следующий пример тестирования программного интерфейса службы `AuthenticationService`:

APP/SHARED/SERVICES/AUTHENTICATION.SERVICE.SPEC.TS

```
import AuthenticationService from './authentication.service';
import {
  describe,
  expect,
  it,
  inject,
  beforeEach,
  beforeEachProviders } from '@angular/core/testing';

describe('shared:AuthenticationService', () => {
  let authenticationService: AuthenticationService;

  beforeEachProviders(() => [
    AuthenticationService
]);

beforeEach(inject(
  [AuthenticationService], (_authenticationService) =>
    {authenticationService = _authenticationService;
  }
));

it('should reject invalid credentials', done => {
  authenticationService.login({
    username: 'foo',
    password: 'bar'})
    .then(success => {
      expect(success).toBeFalsy();
      done();
    });
});

describe('emits an event upon user auth status changes', () => {

  it('that should be truthy for successful logins', done => {
      authenticationService
      .userIsloggedIn
      .subscribe((authStatus: boolean) => {
```

```
    expect(authStatus).toBeTruthy();
    done();
  });

  authenticationService.login({
    username: 'john.doe@mail.com',
    password: 'letmein'
  });
});

it('that should be falsy for failed logins', done => {
    authenticationService
    .userIsloggedIn
    .subscribe((authStatus: boolean) => {
      expect(authStatus).toBeFalsy();
      done();
    });

    authenticationService.login({
      username: 'foo',
      password: 'bar'
    });
});
```

Тест кажется объемным, но на самом деле он довольно прост, поскольку здесь применяется все, что уже известно о модульном тестировании. Единственное, что стоит отметить, – это порядок обертывания утверждений в вызов метода `subscribe()` объекта, возвращаемого вызовом `userIsLoggedIn`, что помогает проверить утверждение только после того, как возникшее событие достигнет функции подписки:

```
authenticationService
  .userIsloggedIn
  .subscribe((authStatus: boolean) => {
    expect(authStatus).toBeTruthy();
    done();
  });
```

Следующий блок выполняет запрос на аутентификацию, поскольку функция `subscribe()` возбудила ожидаемое событие:

```
authenticationService.login({
  username: 'john.doe@mail.com',
  password: 'letmein'
});
```

И последнее, но важное замечание: обратите внимание на вложение одного набора `describe()` в другой. Этот способ часто применяется для группировки тестов по функциональным областям, облегчая отключение тестов при появлении такой необходимости.

Реализация фиктивных Http-ответов с помощью MockBackend

Предыдущий пример является несколько надуманным, поскольку модуль `AuthenticationService` фактически является фикцией, не имеющей реальной реализации. Действующая версия `AuthenticationService` должна реализовать асинхронный метод отправки POST-запроса службе аутентификации. Внесем изменения в текущую реализацию службы и включим эту функцию в виде метода с другим именем, чтобы избежать конфликта с текущей реализацией метода `login()` и его тестами:

APP/SHARED/SERVICES/AUTHENTICATION.SERVICE.TS

```
...
httpLogin(credentials): Promise<boolean> {
  return new Promise(resolve => {
const url = '/api/authentication'; // Либо ваш URL проверки подлинности
const body = JSON.stringify(credentials);
const headers = new Headers({'Content-Type':'application/json'});
const options = new RequestOptions({ headers: headers });

this.http.post(url, body, options)
  .map(response => response.json())
  .subscribe(authResponse => {
      let validCredentials: boolean = false;

      if(authResponse && authResponse.token) {
        validCredentials = true;
        window.sessionStorage.setItem(
          'token',
          authResponse.token
        );
        }

        this.userIsloggedIn.emit(validCredentials);
        resolve(validCredentials);
      },
      error => console.log(error)
    );
  });
}
...
```

Мы добавили в реализацию службы `AuthenticationService` новый асинхронный метод `httpLogin()`, посылающий настоящий HTTP POST-запрос службе аутентификации с учетными данными в формате JSON и возвращающий объект `Promise`, разрешаемый логическим значением после сохранения маркера в локальном хранилище сеанса.



Для обеспечения многократного использования метод должен вернуть HTTP-ответ после его извлечения, что позволит клиентам решить, как сохранить информацию в ответе и что делать дальше. Для краткости оставим этот метод без изменений.

При всех достоинствах новый асинхронный метод влечет следующие проблемы:

- во-первых, следует объявить зависимости, необходимые HTTP-соединениям;
- во-вторых, новый метод выполняет реальный HTTP-запрос к удаленной службе, что выводит его за рамки возможностей тестирования, поэтому такой запрос нужно как-то перехватить и вернуть фиктивный ответ.

Что касается первой проблемы, то в предыдущих разделах уже было описано, как объявлять поставщиков и создавать экземпляры `Http` с помощью инструмента внедрения зависимостей, используя класс `MockBackend` в конструкторе зависимостей.

Однако теперь пришло время использовать все функциональные возможности класса `MockBackend` для перехвата HTTP-соединений и подмены ответов. Вернемся в файл `authentication.service.spec.ts` и заменим текущую реализацию функций `beforeEachProviders()` и `beforeEach()`, попутно импортировав несколько дополнительных и уже знакомых имен:

APP/SHARED/SERVICES/AUTHENTICATION.SERVICE.SPEC.TS (обновленный)

```
import AuthenticationService from './authentication.service';
import { provide } from '@angular/core';
import {
  describe,
  expect,
  it,
  inject,
  beforeEach,
  beforeEachProviders } from '@angular/core/testing';
import {
  Http,
  BaseRequestOptions,
```

```
  Response,
  ResponseOptions } from '@angular/http';
import { MockBackend, MockConnection } from '@angular/http/testing';
import 'rxjs/add/operator/map';

describe('shared:AuthenticationService', () => {
  let authenticationService: AuthenticationService;
  let mockBackend: MockBackend;
  beforeEachProviders(() => [
    MockBackend,
    BaseRequestOptions,
    provide(Http, {
      useFactory: (
        backend: MockBackend,
        options: BaseRequestOptions
      ) => {
        return new Http(backend, options);
      },
      deps: [MockBackend, BaseRequestOptions]
    }),
    AuthenticationService
  ]);

  beforeEach(inject(
    [MockBackend, AuthenticationService],
    (_mockBackend, _authenticationService) => {
      authenticationService = _authenticationService;
      mockBackend = _mockBackend;
    }
  ));

  it('can fetch a valid token when querying the Auth API', done => {
    const mockedResponse = new ResponseOptions({
      body: '{"token": "eyJhbGciOi"}'
    });

mockBackend.connections.subscribe(
(connection: MockConnection) => {
  if(connection.request.url === '/api/authentication') {
    connection.mockRespond(new Response(mockedResponse));
  }
}
);

authenticationService.httpLogin({
  username: 'foo',
```

```
      password: 'bar'
    }).then(success => {
      expect(success).toBeTruthy();
      done();
    },
    error => done.fail(error)
  );
});

// Остальные спецификации тестов без изменений
// ...
```

Сначала мы импортировали все, что требуется для взаимодействия со службами, основанными на HTTP. Реализация функций `beforeEachProviders()` и `beforeEach()` ничем не отличается от той, что мы видели в файле `timer-widget.component.test.ts`. Единственный нюанс, который стоит отметить, – это присваивание нового экземпляра класса `MockBackend` переменной `mockBackend` при каждом выполнении теста. Это необходимо для использования его в новых тестах. Разберем код подробнее. Во-первых, мы сконструировали ответ с фиктивными данными. Этот ответ в дальнейшем будет использован при выполнении реальных HTTP-запросов:

```
const mockedResponse = new ResponseOptions({
  body: '{"token": "eyJhbGciOi"}'
});
```

Объекты `MockBackend` имеют свойство `connections` типа `EventEmitter`, обеспечивающего отправку события объектом `MockConnection` каждый раз, когда он обнаруживает попытку выполнить XHR-подключение через класс `Http` (который для соединения с сервером, как упоминалось, использует класс `mockBackend`). Объект `MockConnection` содержит информацию о попытке запроса, которую можно использовать для нужд тестирования и возвращения конкретного ответа с учетом потребностей. Для этого мы используем метод `mockRespond()` объекта `connection`:

```
mockBackend.connections.subscribe(
  (connection: MockConnection) => {
    if(connection.request.url === '/api/authentication') {
      connection.mockRespond(new Response(mockedResponse));
    }
  }
);
```

При наличии всех этих элементов выполнение реального запроса с помощью методов службы и анализ ответов значительно упрощаются:

```
authenticationService.httpLogin({
  username: 'foo',
  password: 'bar'
}).then(success => {
  expect(success).toBeTruthy();
  done();
},
  error => done.fail(error)
);
```

Тестирование директив

Последний этап путешествия в мир модульного тестирования элементов Angular 2 посвящен директивам. Общая форма директив достаточна проста, поскольку они, по сути, являются компонентами без представлений. Тот факт, что директивы обычно работают с компонентами, является хорошей подсказкой при реализации их тестирования.

Для нужд тестирования можно создать компонент-заглушку, а затем привязать к нему директиву, либо сразу после ее определения, либо с помощью метода `overrideDirectives()` объекта `TestComponentBuilder`, подготовленного для теста. В этом смысле компонент как элемент, содержащий директиву, будет делегировать тестовые операции, поэтому не будем углубляться в этот подход, уже описанный при рассмотрении тестирования компонентов в предыдущих разделах.

Другой подход заключается в привязке обработчиков к директиве и их тестировании, чтобы убедиться в наличии требуемых функциональных возможностей. Рассмотрим пример применения этого подхода для тестирования модуля `TaskTooltipDirective`:

APP/TASKS/TASK-TOOLTIP.DIRECTIVE.SPEC.TS

```
import { Task } from '../shared/shared';
import TaskTooltipDirective from './task-tooltip.directive';
import {
  describe,
  expect,
  it,

beforeEach } from '@angular/core/testing';
```

```
describe('shared:TaskTooltipDirective', () => {
  let taskTooltipDirective: TaskTooltipDirective;

  beforeEach(() => {
    taskTooltipDirective = new TaskTooltipDirective();
  });

  it('should update a given tooltip upon mouseover', done => {
    let mockTooltip = { innerText: '' };
    taskTooltipDirective.task = <Task>{ name: 'Foo' };
    taskTooltipDirective.taskTooltip = mockTooltip;

    taskTooltipDirective.onMouseOver();
    expect(mockTooltip.innerText).toBe('Foo');

    done();
  });

  it('should restore a given tooltip upon mouseout', done => {
    let mockTooltip = { innerText: 'Foo' };
    taskTooltipDirective.task = <Task>{ name: 'Bar' };
    taskTooltipDirective.taskTooltip = mockTooltip;

    taskTooltipDirective.onMouseOver();
    expect(mockTooltip.innerText).toBe('Bar');

    taskTooltipDirective.onMouseOut();
    expect(mockTooltip.innerText).toBe('Foo');

    done();
  });
});
```

Здесь экземпляр объекта директивы создается непосредственно, так как она не имеет зависимостей. Поскольку все операции, выполняемые этой директивой, управляются методами класса, декорированного декоратором `@HostListener()`, необходимо лишь заполнить входные свойства класса директивы фиктивными данными и проверить ожидаемое поведение.

Предстоящий путь

Этот последний пример тестирования завершает знакомство с приемами модульного тестирования в Angular 2, но следует понимать, что это было лишь краткое введение. При тестировании веб-приложений в целом и приложений, использующих Angular 2, в частности возникает масса ситуаций, требующих специфичного подхода. Имейте в виду, что если реализация конкретного теста становится слишком

громоздкой и запутанной, скорее всего, следует переосмыслить код самого тестируемого модуля.

Куда двигаться дальше? Для тех, кто стремится стать великим ниндзя тестирования, имеется несколько путей совершенствования навыков тестирования веб-приложений с использованием Angular 2.

Использование Jasmine в сочетании с Karma

До сих пор для выполнения тестов и получения отчета об их результатах использовался инструмент Jasmine. Такое решение отлично подходит для небольших проектов, но может оказаться нелучшим выбором для тестирования больших проектов, особенно если требуется организовать автоматическое тестирование при изменении кода или обеспечить непрерывную интеграцию тестов с сервером.

Тогда следует перейти на использование более мощных и быстрых средств тестирования без ущерба для проекта. Лучшим выбором для этого станет платформа тестирования Karma. Эта платформа используется командой Angular, она совместима с Jasmine и другими платформами тестирования, такими как Mocha или QUnit, обладает простыми, но очень мощными настройками с поддержкой автоматического сканирования тестов, обеспечивает отслеживание файлов, вывод нескольких отчетов и поддерживает расширение плагинами.

Для непрерывной интеграции с сервером Karma предусматривает адаптеры для Jenkins, Travis и Semaphore.

Более подробную информацию можно найти по адресу: https://karma-runner.github.io.

Получение отчетов об охвате кода тестами

Как узнать, насколько полно тесты охватывают тестируемое приложение? Можно ли быть уверенным, что не осталось непроверенных фрагментов, и если это не так, то каково их соотношение с полной базой кода? Как обнаружить фрагменты кода, которые не охвачены тестированием, чтобы можно было оценить необходимость их тестирования?

Эти проблемы легко решаются путем получения отчета об охвате тестируемого приложения тестами. Инструменты оценки охвата кода отслеживают область модульного тестирования и выдают отчет, сообщающий об общем охвате кода тестами и какие фрагменты остались неохваченными.

Существует несколько инструментов для оценки охвата кода приложений. В настоящее время наиболее популярны Blanket (http://

blanketjs.org) и Istanbul (https://gotwarlost.github.io/istanbul). Они устанавливаются весьма просто и быстро.

Реализация E2E-тестов

В этой главе мы видели, как протестировать определенные части пользовательского интерфейса путем оценки состояния модели DOM. Это позволяет получить представление о том, как будет выглядеть приложение с точки зрения конечного пользователя, но в конечном итоге все основывается лишь на догадках.

Тестирование End-to-end (E2E) – это методология тестирования веб-приложений с помощью автоматизированных агентов, эмулирующих действия пользователя от начала до конца. В отличие от модульного тестирования, здесь не важны нюансы реализации, поскольку E2E-тестирование представляет собой тестирование приложения только с точки зрения пользователя. Такой подход позволяет комплексно проверить работу приложения. В то время как модульное тестирование фокусируется на надежности каждой конкретной части картины, собираемой из фрагментов, E2E-тестирование оценивает картину в целом, выявляя проблемы, возникающие при интеграции компонентов, которые часто упускают из виду модульные тесты.

Команда Angular создала для предыдущей версии Angular мощный инструмент Protractor (https://docs.angularjs.org/guide/e2e-testing), который определила следующим образом:

> «*...инструмент тестирования, имитирующий взаимодействия с пользователем и помогающий проверить качество Angular-приложения*».

Синтаксис тестов для этого инструмента покажется вам знакомым, поскольку для сведения воедино всех тестов он также использует Jasmine. К сожалению, тема E2E-тестирования выходит за рамки этой книги, но имеется несколько источников, которые помогут вам расширить знания в данном вопросе. Одним из них является книга «*Angular 2 Test-driven development*» издательства Packt, содержащая обширные сведения о создании наборов E2E-тестов с использованием инструмента Protractor для приложений на основе Angular 2.

Итоги

Вот и подошло к концу наше путешествие, оно было долгим, но, надеюсь, захватывающим. Эта глава рассказала о важности модульного

тестирования приложений на основе Angular 2, основных формах модульных тестов и процессе настройки Jasmine для выполнения тестов. Также была рассмотрена реализация тестов для компонентов, директив, фильтров, маршрутов и служб. Кроме того, мы обсудили пути дальнейшего освоения Angular 2. Справедливости ради следует отметить, что впереди еще долгий путь, но по-другому и не бывает.

Эта глава завершает книгу, и набираться опыта придется уже за ее пределами. Фреймворк Angular 2 достаточно молодой, и он еще продолжит свое развитие. Надеюсь, что вы станете одним из тех, кто поможет ему в этом. Если у вас получится, дайте знать автору.

Спасибо, что нашли время для этой книги.

Предметный указатель

A

Angular 2
  URL-адрес, 26
  директивы, 110
  зависимости, внедрение по дереву компонентов, 159
  инструмент внедрения зависимостей, расширенная поддержка для пользовательских сущностей, 163
  классы TypeScript, 34
  компиляция TypeScript в понятный браузерам код на JavaScript, 35
  контейнер, 38
  метаданные декораторов, определение, 34
  определение, 23, 33
  приложения, инициализация с помощью функции bootstrap(), 165
  примеры, 39
  ссылки, 170
  функции сопоставления, определение, 319
  шаблоны, редактирование, 41
Angular 2, компоненты
  вывод данных в представление, 53
  методы компонента, 51
  обновление данных, 52
  определение, 48
  улучшение производительности, 48
Angular 2, поддержка маршрутизации, реализация, 211
Angular 2, шпаргалка синтаксических конструкций, URL-адрес, 94
AnimationBuilder
  CssAnimationBuilder, 299
  анимация компонентов, 295
  состояния анимации, отслеживание с помощью класса Animation, 300
AtScript, 79

B

Blanket, URL-адрес, 344
Bootstrap, 29

C

CSS-классы, управление анимацией, 293
CSS-стили
  встроенные таблицы стилей, 106
  инкапсуляция, 105
  свойства стилей, 105
  свойство styleUrls, 106
  управление инкапсуляцией представлений, 107

E

ECMAScript 6 (ES6, ES2015), 59

G

Gulp, URL-адрес, 47

H

HTML-контейнер, построение таблицы списка заданий с помощью директив Angular, 122

J

Jasmine
  URL-адрес, 315
  использование с Karma, 344

JSBIN, URL-адрес, 191

K

Karma, URL-адрес, 344

N

ngAnimate
  описание, 307
  определение, 307
Node.js, URL-адрес, 26

P

Pomodoro Technique,
URL-адрес, 48
Protractor, URL-адрес, 345

R

RAD. *См.* Быстрая разработка приложений

S

SRID. *См.* Идентификатор пространственной привязки

T

TMS-протокол. *См.* Протокол веб-службы сборных цифровых карт
TMS-сервер. *См.* Сервер веб-службы сборных цифровых карт
TypeScript
  URL-адрес, 61
  декораторы, 79
  динамическая типизация
  с помощью типа any, 64
  интерфейсы, 76
  использование других синтаксисов, 25
  область видмости
  с лямбда-функциями, 71
  описание, 25
  определение, 90
  параметры функций, 69
  преимущества, 59
  причины создания, 58
  тип boolean, 64
  тип number, 64
  тип string, 63
  тип void, 66
  типы, 62
TypeScript, ресурсы
  внедрение зависимостей,
  определение, 312
  определение, 61
  официальный сайт, 61

W

WebPack, URL-адрес, 29

А

Анатомия класса, определение, 75
Анимация
  с помощью классов-событий CSS, 293
  с помощью стандартной CSS-разметки, 290
Асинхронная информация
  способы обработки, 189

Б

Библиотека RxJS, 196
Библиотека единой абстрактной модели геоданных GDAL. *См.* GDAL, библиотека

В

Веб-компоненты
  импорт HTML, 24
  определение, 24
  пользовательские элементы, 24
  теневая модель DOM, 24
  шаблоны, 24
Внедрение зависимостей, 152
  ограничение, 160
Входные данные, проверка допустимости, 251
Высококачественная глобальная цифровая модель местности с базой в 1 км на пиксель. *См.* GLOBE, проект

Г

Геопространственные данные.
См. Геоданные
Глобальная последовательная иерархическая географическая база данных высокого разрешения. *См.* GSHHG, географическая база данных
Группа элементов управления
  описание, 255
  определение, 262

Д

Двухсторонняя привязка данных
  директива NgModel, 247
  описание, 244
Декораторы TypeScript
  декораторы класов, 80
  декораторы методов, 83
  декораторы параметров, 86
  декораторы свойств, 81
Декораторы классов, 79
Дерево компонентов, определение, 148
Директива ngControl, создание элементов управления в модели DOM, 257
Директива NgControlGroup
  группировка элементов управления в модели DOM, 258
Директива NgFor, 112
Директива NgIf, 111
Директивы
  встроенные, 111
  директива NgClass, 113
  директива NgStyle, 112
  директива NgSwitch, 113
  директива NgSwitchDefault, 114
  директива NgSwitchWhen, 113
  описание, 110
  тестирование, 342
Документация Angular, URL-адрес, 202
Дочерние маршрутизаторы
  определение, 230
  ссылки на дочерние маршруты, 230

З

Зависимости, 152

И

Именованные ключевые кадры, 291
Инкапсуляция представлений, 107
Интегрированные среды разработки
  Atom, 44
  Sublime Text 3, 43
  Visual Studio Code, 45
  WebStorm, 46
  использование Gulp с другими интегрированными средами разработки, 46
  усовершенствование, 43
Интерактивность элементов управления, отслеживание, 251
Интерактивность элементов управления, отслеживание изменений с помощью локальных ссылок, 254
Интерфейс класса
  описание, 73
  расширение, 78
Интерфейсы, 73

К

Класс FormBuilder, 255
Класс Headers, URL-адрес, 201
Класс Title, URL-адрес, 233
Класс Validator, 256
Классы, 73
Классы-события
  описание, 295
  определение, 295
Клиентская служба проверки подлинности
  блокировка несанкционированного доступа, 277
  импортирование в других компонентах, 276

отражение состояния
на пользовательском
интерфейсе, 280
эмуляция, 275
Компонент входа
контекст фунции входа, 263
мониторинг изменения состояния
элементов управления, 270
пользовательская директива
безопасности RouterOutlet,
разработка, 282
пользовательская проверка
допустимости, 268
разработка, 263
реализация, 266
управление доступом, 281
шаблон функции входа, 266
Компоненты
контекст заданий, 184
контекст таймера, 177
корневой компонент,
определение, 185
переопределение зависимостей
для улучшения тестирования, 330
свойства и методы, 322
создание, 177
тестирование
с зависимостями, 326

Л

Ловушки цикла существования
маршрутизатора
асинхронная загрузка
компонентов с помощью
AsyncRoutes, 240
дополнительные советы
и хитрости, 237
настройка генерируемых
URL-адресов с учетом стратегии
размещения, 239
переадресация на другие
маршруты, 237
точная настройка базового
пути, 238
Локальные ссылки, отслеживание
изменений, 254
Лямбда-функции, 67

М

Маршруты
тестирование, 332
тестирование переадресации, 334
тестирование с помощью
URL-адреса, 333
Массивы, 64
Масштабируемые приложения
общие соглашения, 149
соглашение об именовании
файлов и модулей, 151
Международная ассоциация
производителей нефти и газа.
См. Европейская нефтяная
топографическая группа (EPSG)
Методы bootstrap, действия,
определение, 41

Н

Наблюдаемые объекты,
введение, 191

О

Общий конекст, настройка
параметров приложения
из центральной службы, 174
Общий контекст
определение, 170
службы, 173
Оператор класса, элементы, 74
Отслеживание данных, добавление
новых заданий, 208
Отслеживание данных
по HTTP, 204
Отчет об охвате тестами,
определение, 344
Официальное хранилище модулей
NPM, ссылки, 40

П

Параметры маршрутизации
динамическая передача, 223

обработка, 222
разбор с помощью службы RouteParams, 225
Параметры функций в TypeScript
необязательные параметры, 69
оставшиеся параметры, 70
параметры по умолчанию, 69
перегрузка функций, 70
Поиск поставщиков, ограничение, 160
Полигон. *См.* Многоугольник
Полилиния. *См.* Ломаная линия
Пользовательские анимационные директивы
взаимодействие с директивами шаблона, 305
разработка, 301
Пользовательские директивы
соглашение об именах, 145
создание, 140
создание подсказок с помощью пользовательской директивы, 143
Пользовательские значения, настройка, 96
Пользовательские события, передача данных между компонентами, 100
Пользовательские фильтры
анатомия, 136
фильтрация данных, 139
форматирование времени, 137
Пользовательские элементы, присваивание имен, 49
Поток выполнения, 67
Привязки Python. *См.* Обертка Python
Прикладной программный интерфейс HTTP
внедрение класса http, 202
внедрение символа модулей HTTP_PROVIDERS, 202
использование класса Request, 200
использование класса RequestOptionsArgs, 200
обработка ошибок при выполнении http-запросов, 202
объект Response, 201
описание, 199
Приложение, начальная загрузка, 186
Приложение Angular 2, модификация, 170
Приложение по идентификации береговых линий, городов и озер в заданном радиусе. *См.* DISTAL, веб-приложение
Приложение с bootstrap(), обнаружения изменений встроенным профилировщиком, 166
Приложение с модулями
внешние модули, 89
внутренние модули, 87
организация, 87
Приложения с bootstrap()
определение, 165
переключение между режимами разработки и эксплуатации, 166
Принцип персональной ответственности, 170
Проекции. *См.* Картографические проекции
Протоколы. *См.* Геопространственные веб-протоколы

Р

Рабочая область
настройка, 26
установка зависимостей, 26
установка типирования TypeScript, 31
Реактивное функциональное программирование
библиотека RxJS, 196
в Angular 2, 193

С

Сигнатуры функций, расширение декоратором класса, 81

Синтаксис шаблонов
альтернативный синтаксис для входных и выходных свойств, 103
взаимодействие через пользовательские события, 98
входные и выходные свойства, 95
локальные ссылки в шаблонах, 102
описание, 92
привязка выражений, 94
привязка данных с помощью входных свойств, 93
привязка событий с помощью выходных свойств, 94
Система топологически интегрированной географической кодировки и привязки. *См.* TIGER, база данных
Службы
асинхронные службы, тестирование, 335
реализация фиктивных http-ответов с помощью MockBackend, 338
тестирование, 334
Службы маршрутизации
CSS-ловушки для активных маршрутов, 222
директивы маршрутизатора, определение, 218
настройка декоратора RouteConfig с помощью экземпляров RouteDefinition, 216
принудительный переход по маршрутам, 220
создание нового компонента для демонстрационных целей, 214
Событие маршрутизатора CanDeactivate, обход при отправке формы, 250
События жизненного цикла маршрутизатора
CanActivate, 231
CanDeactivate, 234
CanReuse, 235
OnActivate, 233
OnDeactivate, 234
OnReuse, 235
описание, 230
Соглашение о присвоении, обеспечение непрерывной масштабируемости с помощью фасадов или баррелей, 152
Спецификации CSS, URL-адрес, 107
Список заданий в проекте Pomodoro
HTML-контейнер, настройка, 121
описание, 120
Список заданий проекта Pomodoro
встраивание дочерних компонентов, 132
отображение изменения состояния в шаблонах, 130
переключение заданий, 128
Среда тестирования
настройка, 315
настройка команд NPM, 318
реализация средства выполнения тестов, 315
Статические методы проверки допустимости
composeAsync(), 257
compose(validators Function[]), 257
maxLength(maxLength number), 256
minLength(minLength number), 256
pattern(pattern string), 256
required, 256
описание, 256
Страница Playground, URL-адрес, 75
Структура каталогов приложения Pomodoro, описание, 167

T

Тестирование маршрутов, 332

Тесты, отладка, 325
Тесты E2E
  описание, 345
  реализация, 345
Тип string
  объявление переменных, 63
  описание, 63
Тип перечисления, 65
Трансформируемые свойства, 291

У

Универсальные шаблоны, ссылки, 85
Уровень представления. *См.* Слой клиента

Ф

Фильтры
  async, 120
  currency, 116
  date, 117
  i18nPlural, 119
  i18nSelect, 119
  JSON, 118
  number, 116
  percent, 116
  replace, 118
  slice, 117
  uppercase/lowercase, 115
  интернационализации, 119
  описание, 114
  соглашения об именах, 145
  создание, 136
  тестирование, 320
  управление привязками шаблонов, 114
Форма, связанная с директивой NgModel, 247
Функции
  аннотации типов, 67
  описание, 67

Ш

Шаблоны
  внешние, 104
  внутренние, 104
  настройка из компонентов, 103
  управление привязками с помощью фильтров, 114

Э

Элементы управления
  группировка в модели DOM с помощью директивы NgControlGroup, 258
  описание, 255
  Соединение модели DOM и контроллера с помощью ngFormModel, 262
  создание в модели DOM с помощью директивы ngControl, 257

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс»
наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Пабло Дилеман

Изучаем Angular 2

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com

Перевод *Рагимов А. Н.*
Научный редактор *Киселев А. Н.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 60×90 1/16.
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 22,12. Тираж 200 экз.

Веб-сайт издательства: www.дмк.рф

Фреймворк Angular 2 полностью переписан с нуля для удовлетворения требований современных разработчиков, которым необходима высокая производительность и отзывчивость веб-приложений.

Эта книга поможет вам освоить основы проектирования и разработки компонентов Angular 2, обеспечивая полный охват синтаксиса языка TypeScript, знание которого необходимо для следования за примерами кода. После создания первых компонентов мы объединим их в полноценное веб-приложение. Затем перейдем к реализации маршрутов в Angular 2, узнаем, как организовать поддержку разных состояний приложения и навигацию между компонентами. После этого мы займемся веб-формами и проверкой ввода пользователя, а затем используем новые знания как фундамент для изучения основ аутентификации пользователей и различных способов защиты страниц и разделов сайта. Так же в этой книге рассматривается тема анимации компонентов и элементов DOM. В заключительной части дается детальное освещение приемов модульного тестирования, не только компонентов, но и других элементов, таких как службы, директивы, маршруты и фильтры.

Чему вы научитесь, прочитав эту книгу:

- настраивать рабочую среду с полным набором инструментов, необходимых для создания компонентов Angular 2, с минимальными усилиями;
- программировать на TypeScript – мощном типизированном надмножестве JavaScript, поддерживающем компиляцию в обычный код на JavaScript;
- управлять отображением данных и их обновлением при изменении;
- создавать мощные веб-приложения с иерархической структурой компонентов, способные генерировать и обрабатывать события, распространяемые по всему дереву элементов;
- пользоваться внешними библиотеками и службами, а также мощными функциями редактирования, заключенными в веб-формах Angular 2;
- осуществлять навигацию между компонентами и обрабатывать различные состояния приложения;
- разрабатывать надежные приложения с применением технологий модульного тестирования и инструментов отладки.



Кому адресована эта книга

Эта книга адресована веб-разработчикам, которые хотят освоить разработку современных мобильных и настольных веб-приложений следующего поколения с помощью Angular 2. От читателя не требуется опыт работы с Angular 1.x или 2, но предполагается уверенное владение языком JavaScript. Она отлично подойдет новичкам в Angular, нуждающимся в пояснении и определении его концепций.

ISBN 978-5-97060-461-8

ДМК ПРЕСС ИЗДАТЕЛЬСТВО
www.дмк.рф

Интернет-магазин www.dmkpress.com
Книга-почтой: orders@alians-kniga.ru
Оптовая продажа: «Альянс-книга»
(499)782-3889, books@alians-kniga.ru