Angular и TypeScript

Сайтостроение для профессионалов

Яков Файн
Антон Моисеев

MANNING

Angular 2 Development with TypeScript

YAKOV FAIN
ANTON MOISEEV

MANNING
Shelter Island

Яков Файн, Антон Моисеев

Angular и TypeScript

Сайтостроение для профессионалов

ПИТЕР®
Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону • Самара • Минск
2018

Яков Файн, Антон Моисеев

Angular и TypeScript. Сайтостроение для профессионалов

Серия «Библиотека программиста»

Перевели с английского *Н. Вильчинский, Е. Зазноба*

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*О. Сивченко*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Н. Хлебина*
Художественный редактор	*С. Заматевская*
Корректоры	*Е. Павлович, Е. Рафалюк-Бузовская*
Верстка	*Г. Блинов*

ББК 32.988.02
УДК 004.738.5

Файн Я., Моисеев А.

Ф17 Angular и TypeScript. Сайтостроение для профессионалов. — СПб.: Питер, 2018. — 464 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-0496-3

Если вы занимаетесь веб-разработкой — от веб-клиентов до полнофункциональных одностраничных приложений, — то фреймворк Angular будет для вас просто спасением. Этот ультрасовременный инструмент полностью интегрирован со статически типизированным языком TypeScript, который отлично вписывается в экосистему JavaScript.

Вы научитесь:

- Проектировать и строить модульные приложения.
- Правильно транспилировать TypeScript в JavaScript.
- Пользоваться новейшими инструментами JavaScript — в частности npm, Karma и Webpack.

Если вам знаком язык JavaScript — берите и читайте! Знаний TypeScript или AngularJS для изучения книги не требуется.

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

ISBN 978-1617293122 англ.
ISBN 978-5-4461-0496-3

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 191123, Россия, город Санкт-Петербург,
улица Радищева, дом 39, корпус Д, офис 415. Тел.: +78127037373.
Дата изготовления: 12.2017. Наименование: книжная продукция. Срок годности: не ограничен.
Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —
Книги печатные профессиональные, технические и научные.
Подписано в печать 08.12.17. Формат 70×100/16. Бумага офсетная. Усл. п. л. 37,410. Тираж 1000. Заказ 0000.
Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87

Краткое содержание

Оглавление

Предисловие

Мы принялись подбирать хороший фреймворк для JavaScript примерно четыре года назад. Тогда мы работали над платформой для страховой компании и большая часть интерфейса системы была написана с помощью фреймворка Apache Flex (ранее известного как Adobe Flex). Это отличный инструмент для разработки веб-интерфейсов, но он требует наличия Flash Player, с некоторых пор впавшего в немилость.

После попыток создать несколько вводных проектов, написанных на JavaScript, мы обнаружили заметное снижение производительности наших разработчиков. Задача, выполнить которую с использованием Flex можно было за один день, требовала три дня работы в других фреймворках JavaScript, включая AngularJS. Основная причина спада — нехватка типов в JavaScript, недостаточная поддержка IDE и отсутствие поддержки компилятора.

Когда мы узнали, что компания Google начала разработку фреймворка Angular с TypeScript в качестве рекомендуемого языка, мы взяли данный фреймворк на вооружение летом 2015 г. В то время о нем было известно немногое. В нашем распоряжении была лишь информация из некоторых блогов и видеороликов, записанных на конференции, где команда разработчиков Angular представляла новый фреймворк. Нашим основным источником информации был его исходный код. Но вскоре мы обнаружили у Angular отличный потенциал, поэтому решили начать создавать с его помощью обучающее ПО для отдела разработки нашей компании. В то же время Майк Стивенс (Mike Stephens), сотрудник издательства Manning, искал авторов, заинтересованных в написании книги об Angular. Именно так и появилось это издание.

После года работы с Angular и TypeScript мы можем подтвердить: этот дуэт предоставляет наиболее продуктивный способ создания средних и крупных веб-приложений, которые могут работать в любом современном браузере, а также на мобильных платформах. Перечислим основные причины, позволившие нам прийти к такому выводу.

- *Четкое разделение интерфейса и логики.* Код, который отвечает за пользовательский интерфейс, и код, реализующий логику приложения, четко разделены. Интерфейс не должен создаваться в HTML, существуют другие продукты с поддержкой его нативной отрисовки для iOS и Android.
- *Модульность.* Существует простой механизм разбиения приложения на модули с возможностью их ленивой загрузки.
- *Поддержка навигации.* Маршрутизатор поддерживает сложные сценарии навигации в одностраничных приложениях.
- *Слабое связывание.* Внедрение зависимостей (Dependency Injection, DI) позволяет установить связь между компонентами и сервисами. С помощью связывания

и событий есть возможность создавать слабо связанные компоненты, которые можно использовать повторно.

- *Жизненный цикл компонентов*. Каждый компонент проходит через тщательно определенный жизненный цикл, разработчикам доступны процедуры, позволяющие перехватывать важные события компонентов.
- *Определение изменений*. Автоматический (и быстрый) механизм определения изменений дает возможность отказаться от обновления интерфейса вручную, также предоставляя способ выполнить тонкую настройку этого процесса.
- *Отсутствие ада обратных вызовов*. Angular поставляется вместе с библиотекой RxJS, которая позволяет построить процесс обработки асинхронных данных, основанный на подписке, что помогает избежать появления ада обратных вызовов.
- *Формы и валидация*. Поддержка форм и пользовательская валидация хорошо спроектированы. Вы можете создавать формы, добавляя директивы к элементам формы как в шаблонах, так и программно.
- *Тестирование*. Фреймворк поддерживает модульное (блочное) и сквозное тестирование, вы можете интегрировать тесты в свой автоматизированный процесс сборки.
- *Упаковка и оптимизация с помощью Webpack*. Упаковка и оптимизация кода с помощью Webpack (и различных плагинов для него) позволяет поддерживать небольшой размер развертываемых приложений.
- *Инструментальные средства*. Инструментальные средства поддерживаются так же хорошо, как и для платформ Java и .NET. Анализатор кода TypeScript указывает на ошибки по мере набора текста, а инструмент для создания временных платформ и развертывания (Angular CLI) позволяет избегать написания стереотипного кода и сценариев конфигурирования.
- *Лаконичный код*. Использование классов и интерфейсов TypeScript делает код более лаконичным, а также упрощает его чтение и написание.
- *Компиляторы*. Компилятор TypeScript генерирует код JavaScript, который человек может прочитать. Код TypeScript может быть скомпилирован для версий JavaScript ES3, ES5 или ES6. Компиляция перед выполнением (Ahead-of-time, AoT) кода Angular (не путать с компилированием TypeScript) помогает избавиться от необходимости поставлять с вашим приложением компилятор для Angular, что еще больше снижает накладные расходы, требуемые для фреймворка.
- *Отрисовка на стороне сервера*. Angular Universal превращает ваше приложение в HTML на этапе сборки кода офлайн. Данный код может быть использован для отрисовки на стороне сервера, что, в свою очередь, значительно улучшает индексирование поисковыми системами и SEO.

Таким образом, Angular 2 готов к работе сразу после установки.

С точки зрения управления этот фреймворк может быть привлекательным, поскольку в мире более миллиона разработчиков предпочитают AngularJS и большинство из них переключатся на Angular. Эта версия была выпущена в сентябре

2016 г., и каждые полгода будут выпускаться новые крупные релизы. Команда разработчиков Angular потратила два года на разработку Angular 2.0, и к моменту выхода фреймворка более полумиллиона разработчиков уже использовали его. Наличие большого количества работников с определенными навыками — важный фактор, который следует учитывать при выборе технологий для новых проектов. Кроме того, платформами Java или .NET пользуются более 15 миллионов разработчиков, и многие из них сочтут синтаксис TypeScript гораздо более привлекательным, чем синтаксис JavaScript, благодаря поддержке классов, интерфейсов, обобщенных типов, аннотаций, переменных членов класса, а также закрытых и открытых переменных, не говоря о полезном компиляторе и поддержке известных IDE.

Писать книгу об Angular было трудно, поскольку мы начали работу с ранних альфа-версий фреймворка, которые изменились по мере перехода к бета- и релиз-версиям. Но нам нравится результат, и мы уже начали разрабатывать с помощью Angular проекты для решения задач реальной сложности.

Предисловие к русскому изданию

В эпоху бурного развития Интернета требования к сложности и качеству сайтов постоянно растут. Пользователям и организациям уже недостаточно простых статических страниц. В связи с этим давно наметился спрос на полноценные веб-приложения, однако до недавнего времени он удовлетворялся в основном за счет таких закрытых технологий, как Adobe Flex и JavaFX.

Около десяти лет назад ситуация начала меняться. Последовательная стандартизация языка JavaScript привела к появлению множества инструментов для веб-разработки, которые не требуют дополнений и поддерживаются во всех современных браузерах. Пожалуй, самым известным из них стал фреймворк Angular.

Angular разрабатывается компанией Google и при этом является полностью бесплатным и открытым. Это устоявшийся программный продукт, который пользуется большой популярностью как на корпоративном рынке, так и в небольших студиях. Количество программистов, работающих с ним, давно перевалило за миллион.

Эта книга посвящена Angular 2 — современной версии данного фреймворка. Он вобрал в себя все последние веяния в мире веб-разработки, сохранив традиционные черты, характерные для первой версии. Главной особенностью Angular 2 является его изначальная ориентированность на TypeScript. И хотя поддержка JavaScript (ES5 и ES6) по-прежнему присутствует, TypeScript остается основным языком разработки. Это, в сущности, надстройка над ECMAScript 6 со строгой типизацией. Развитая система типов и программных интерфейсов позволяет писать более безопасный и выразительный код, который легче сопровождать. Кроме того, TypeScript имеет отличную поддержку в современных средах разработки.

Еще одной ключевой особенностью Angular 2 является четкое разделение интерфейса и бизнес-логики. Это позволяет создавать компоненты, которые могут работать в том числе на таких мобильных операционных системах, как iOS и Android. Angular 2 позволяет разбивать код на модули, которые загружаются по мере необходимости и содержат описание своих зависимостей. Это значительно упрощает процедуру тестирования (как модульного, так и сквозного) и облегчает интеграцию тестов в процесс сборки.

Разработчикам доступен богатый инструментарий для написания и развертывания приложений. Автодополнение кода, упаковка и оптимизация, отрисовка на стороне сервера, компиляция (статическая и на лету) — все это становится возможным благодаря современным технологиям, на которых основаны Angular 2 и TypeScript. В частности, для управления проектами предусмотрена утилита командной строки Angular CLI, а для создания гибких пользовательских интерфейсов компания Google предоставляет библиотеку компонентов Angular Material 2.

На протяжении двух лет разработкой Angular 2 занимались сотни программистов. Все старания были направлены на то, чтобы вы получили современный инструмент, полностью готовый к работе сразу после установки. Этот фреймворк покрывает собой все аспекты создания сложных веб-приложений, начиная с обработки событий и навигации и заканчивая развертыванием в промышленных масштабах. Все это и многое другое подробно рассматривается на страницах книги.

Благодарности

Оба автора хотели бы поблагодарить издательство Manning Publications и Алана М. Кауниота (Alain M. Couniot) за его предложения по улучшению технической стороны книги и Коди Сэнда (Cody Sand) — за техническую редактуру. Благодарим также следующих рецензентов, предоставивших ценную обратную связь: Криса Коппенбаргера (Chris Coppenbarger), Дэвида Баркола (David Barkol), Дэвида Ди Мария (David DiMaria), Фредрика Энгберга (Fredrik Engberg), Ирака Илиша Рамоса Эрнандеса (Irach Ilish Ramos Hernandez), Джереми Брайана (Jeremy Bryan), Камала Раджа (Kamal Raj), Лори Уилкинс (Lori Wilkins), Мауро Керциоли (Mauro Quercioli), Себастьяна Нишеля (Sébastien Nichèle), Полину Кесельман (Polina Keselman), Субира Растоги (Subir Rastogi), Свена Лесекана (Sven Lösekann) и Висама Цагала (Wisam Zaghal).

Яков хотел бы поблагодарить своего лучшего друга Сэмми (Sammy) за создание теплой и комфортной атмосферы во время работы над этой книгой. К сожалению, Сэмми не умеет разговаривать, но, безусловно, любит Якова. Порода Сэмми — золотистый ретривер.

Антон хотел бы поблагодарить Якова Файна (Yakov Fain) и издательство Manning Publications за полученную возможность стать соавтором книги и приобрести бесценный писательский опыт. Он также благодарен своей семье за терпение, которое она проявляли, пока он работал над книгой.

Об этой книге

Программы на Angular могут быть написаны с помощью двух версий JavaScript (ES5 и ES6), а также на языках Dart или TypeScript. Сам фреймворк разработан с использованием TypeScript, и именно его мы будем задействовать во всех примерах нашей книги. В приложении Б вы найдете раздел «Зачем создавать Angular-приложения[1] на TypeScript», где мы объясняем причины, по которым выбрали этот язык.

Поскольку мы по своей природе практики, то и книгу писали для практиков. Мы не только объясним особенности фреймворка, приводя простые примеры кода, но и на протяжении нескольких глав покажем, как создать одностраничное приложение для онлайн-аукционов.

Во время написания и редактирования данной книги мы провели несколько семинаров с использованием примеров кода. Это позволило получить раннюю (и крайне положительную) обратную связь об изложенном материале. Мы очень надеемся на то, что вам понравится изучать Angular.

Как читать книгу

Ранние версии книги начинались с глав, посвященных ECMAScript 6 и TypeScript. Некоторые редакторы предложили переместить этот материал в приложения, чтобы читатели смогли быстрее приступить к изучению Angular. Мы согласились с таким изменением. Но если вы еще не знакомы с синтаксисом ECMAScript 6 и TypeScript, то информация, представленная в приложениях, поможет разобраться в примерах кода каждой главы.

Структура издания

Книга состоит из десяти глав и двух приложений.

В главе 1 приводится обзор архитектуры Angular, кратко рассматриваются популярные фреймворки и библиотеки JavaScript. Вы также познакомитесь с приложением для онлайн-аукционов, которое начнете разрабатывать в главе 2.

Вы будете разрабатывать эту программу с помощью TypeScript. Информация, изложенная в приложении Б, позволит приступить к работе с этим замечательным языком, представляющим собой расширенный набор функций языка JavaScript. Вы узнаете не только о том, как писать классы, интерфейсы и обобщенные классы, но и как скомпилировать код TypeScript для современных версий JavaScript, которые могут быть развернуты во всех браузерах. В TypeScript

[1] Здесь и далее: имеется в виду приложение, созданное с помощью Angular. — *Примеч. ред.*

реализована большая часть синтаксиса спецификации ECMAScript 6 (она рассматривается в приложении А), а также синтаксис, который будет включен в будущие релизы ECMAScript.

В главе 2 вы начнете разрабатывать простые приложения с помощью Angular и создадите свои первые Angular-компоненты. Вы узнаете, как работать с загрузчиком модулей SystemJS, и мы предложим вам свою версию стартового проекта Angular, который может быть использован в качестве отправной точки для всех примеров приложений этой книги. В конце главы вы создадите первую версию главной страницы для онлайн-аукциона.

Глава 3 посвящена маршрутизатору Angular, предлагающему гибкий способ настройки навигации в одностраничных приложениях. Вы узнаете, как сконфигурировать маршруты для компонентов-предков и компонентов-потомков, передавать данные из одного маршрута в другой и выполнять «ленивую» загрузку модулей. В конце главы вы проведете рефакторинг для онлайн-аукциона, разбив его на несколько элементов и добавив для них маршрутизацию.

В главе 4 представлена информация о шаблоне (паттерне) «Внедрение зависимостей» и его реализации в Angular. Вы познакомитесь с концепцией поставщиков, которая позволит указать, как создавать объекты внедряемых зависимостей. В новой версии онлайн-аукциона вы примените внедрение зависимостей для наполнения данными представления Product Details (Информация о продукте).

В главе 5 мы рассмотрим разные виды привязки данных, познакомим вас с реактивным программированием с помощью наблюдаемых потоков данных и покажем, как работать с каналами. Глава заканчивается новой версией онлайн-аукциона: в нее добавлен наблюдаемый поток событий, используемый для фильтрации популярных продуктов на главной странице.

Глава 6 посвящена разработке компонентов, которые могут общаться друг с другом в слабо связанной манере. Мы рассмотрим их входные и выходные параметры, шаблон проектирования «Посредник», а также жизненный цикл компонента. Кроме того, в этой главе приводится обзор механизма обнаружения изменений, представленного в Angular. Онлайн-аукцион приобретет функциональность оценки продуктов.

Глава 7 посвящена обработке форм в Angular. После рассмотрения основ Forms API мы обсудим валидацию форм и реализуем ее в поисковом компоненте для еще одной версии онлайн-аукциона.

В главе 8 мы объясним, как клиентское Angular-приложение может взаимодействовать с серверами, используя протоколы HTTP и WebSocket, и рассмотрим примеры. Вы создадите серверное приложение, применяя фреймворки Node.js и Express. Далее вы развернете фрагмент онлайн-аукциона, написанный на Angular, на сервере Node. С помощью клиентской части (front end) аукциона можно будет начать общаться с сервером Node.js с применением протоколов HTTP и WebSocket.

Глава 9 посвящена модульному тестированию. Мы рассмотрим основные принципы работы с Jasmine и библиотекой для проверки Angular. Вы узнаете, как тестировать сервисы, компоненты и маршрутизатор, сконфигурировать и использовать

Karma для запуска тестов, а также реализуете несколько модульных тестов для онлайн-аукциона.

Глава 10 посвящена автоматизации процессов сборки и развертывания. Вы увидите, как можно применять упаковщик Webpack для минимизации и упаковки вашего кода для развертывания.

Кроме того, вы узнаете, как использовать Angular CLI для генерации и развертывания проектов. Размер развернутой версии онлайн-аукциона снизится с 5,5 Мбайт (в разработке) до 350 Кбайт (на производстве).

В приложении А вы познакомитесь с новым синтаксисом, добавленным в ECMAScript 2015 (также известным как ES6). Приложение Б содержит вводную информацию о языке TypeScript.

Соглашения, принятые в этой книге, и материалы для скачивания

В данной книге содержится множество примеров исходного кода, который отформатирован с использованием `моноширинного шрифта`, чтобы выделить его на фоне обычного текста. Нередко оригинальный исходный код переформатирован; добавлены разрывы строк, а также переработаны отступы с учетом свободного места на страницах. Иногда, когда этого оказалось недостаточно, листинги содержат метки продолжения строки (➥). Кроме того, комментарии к исходному коду были удалены там, где код описан в тексте. Во многих листингах приведены комментарии, указывающие на важные концепции.

Исходный код примеров книги доступен для загрузки с сайта издателя https://www.manning.com/books/angular-2-development-with-typescript.

Авторы также создали репозиторий на GitHub, в котором содержатся примеры исходного кода, на https://github.com/Farata/angular2typescript. Если в результате выхода новых релизов Angular какие-то примеры перестанут работать, то на указанной странице вы можете рассказать об этом и авторы постараются разобраться с проблемой.

Об авторах

Яков Файн (Yakov Fain) является сооснователем двух компаний: Farata Systems и SuranceBay. Farata Systems — фирма, занимающаяся IT-консалтингом. Яков руководит отделом обучения и проводит семинары по Angular и Java по всему миру. SuranceBay — производственная компания, которая автоматизирует различные рабочие потоки в отрасли страхования в Соединенных Штатах, предлагая приложения, использующие ПО как услугу (Software as a Service, SaaS). В этой компании Яков руководит различными проектами.

Яков — большой энтузиаст Java. Он написал множество книг, посвященных разработке ПО, а также более тысячи статей в блоге по адресу yakovfain.com. Кроме того, его книга *Java Programming for Kids, Parents and Grandparents* («Программирование на Java для детей, родителей, бабушек и дедушек») доступна для бесплатного скачивания на нескольких языках по адресу http://myflex.org/books/java4kids/java4kids.htm. Его никнейм в Twitter — @yfain.

Антон Моисеев (Anton Moiseev) — ведущий разработчик ПО в компании SuranceBay. Он более десяти лет занимается разработкой промышленных приложений с помощью технологий Java и .NET и имеет богатый опыт создания многофункциональных интернет-приложений для разных платформ. Сфера деятельности Антона — веб-технологии, причем он специализируется на приемах, позволяющих фронтенду и бэкенду без проблем взаимодействовать друг с другом. Он провел множество занятий, посвященных фреймворкам AngularJS и Angular 2.

Время от времени Антон пишет статьи в блоге по адресу antonmoiseev.com. Его никнейм в Twitter — @anton-moiseev.

1 Знакомство с Angular

В этой главе:

- краткий обзор фреймворков и библиотек JavaScript;
- общий обзор Angular 1 и 2;
- набор инструментов для Angular-разработчика;
- пример приложения.

Angular 2 — фреймворк с открытым исходным кодом, написанный на JavaScript и поддерживаемый компанией Google. Он представляет собой полностью переработанную версию своего популярного предшественника, AngularJS. С помощью Angular вы можете разрабатывать приложения на JavaScript (применяя синтаксис ECMAScript 5 или 6), Dart или TypeScript. В книге мы будем использовать TypeScript. Причины, по которым был выбран именно этот синтаксис, описаны в приложении Б.

ПРИМЕЧАНИЕ

Мы не ждем от вас опыта работы с AngularJS, но рассчитываем, что вы знакомы с синтаксисом JavaScript и HTML, а также понимаете, из чего состоит веб-приложение. Кроме того, предполагается, что вы имеете представление о CSS и знакомы с ролью объекта DOM в браузере.

Мы начнем эту главу с очень краткого обзора нескольких популярных фреймворков JavaScript. Далее рассмотрим архитектуру AngularJS и Angular 2, выделяя улучшения, привнесенные в новую версию. Кроме того, кратко рассмотрим инструменты, используемые Angular-разработчиками. Наконец, мы взглянем на приложение-пример, которое будем создавать на протяжении книги.

ПРИМЕЧАНИЕ

Наша книга посвящена фреймворку Angular 2, и для краткости мы будем называть его Angular. Если мы упомянем AngularJS, то это значит, что речь идет о версиях 1.x данного фреймворка.

1.1. Примеры фреймворков и библиотек JavaScript

Обязательно ли использовать фреймворки? Нет, можно написать клиентскую часть веб-приложений на чистом JavaScript. В этом случае не нужно изучать что-то новое, достаточно знания языка JavaScript. Отказ от фреймворков приведет к возникновению трудностей при поддержке совместимости между браузерами, а также к увеличению циклов разработки. Фреймворки же позволяют полностью управлять архитектурой, шаблонами проектирования и стилями кода вашего приложения. Большая часть современных веб-приложений написаны при сочетании нескольких фреймворков и библиотек.

В этом разделе мы кратко рассмотрим популярные фреймворки и библиотеки для работы с JavaScript. В чем заключается разница между ними? *Фреймворки* позволяют структурировать ваш код и заставляют писать его определенным способом. *Библиотеки* обычно предлагают несколько компонентов и API, которые могут быть использованы по желанию в любом коде. Другими словами, фреймворки предоставляют большую гибкость при разработке приложения.

Angular — один из многих фреймворков, применяемых для разработки веб-приложений.

1.1.1. Полноценные фреймворки

Содержат все, что понадобится для разработки веб-приложения. Они предоставляют возможность легко структурировать ваш код и поставляются вместе с библиотекой, содержащей компоненты и инструменты для сборки и развертывания приложения.

Например, *Ext JS* — полноценный фреймворк, созданный и поддерживаемый компанией Sencha. Он поставляется вместе с полным набором UI-компонентов, включающим продвинутые сетки данных и таблицы (их наличие критически важно для разработки промышленных приложений для офисов). Ext JS значительно увеличивает объем кода программы — вам не удастся найти созданное с его помощью приложение, которое весит меньше 1 Мбайт. Кроме того, данный фреймворк довольно глубоко внедряется — будет трудно при необходимости переключиться на другой инструмент.

Sencha также предлагает фреймворк Sencha Touch, используемый при создании веб-приложений для мобильных устройств.

1.1.2. Легковесные фреймворки

Позволяют структурировать веб-приложение, предлагают способ настройки навигации между представлениями, а также разбивают приложения на слои, реализуя шаблон проектирования «Модель — Представление — Контроллер» (Model — View — Controller, MVC). Кроме того, существует группа легковесных фреймворков, которые специализируются на тестировании приложений, написанных на JavaScript.

Angular — фреймворк с открытым исходным кодом, предназначенный для разработки веб-приложений. Упрощает создание пользовательских компонентов, которые могут быть добавлены в документы HTML, а также реализацию логики приложения. Активно использует привязку данных, содержит модуль внедрения зависимостей, поддерживает модульность и предоставляет механизм для настройки маршрутизации. AngularJS был основан на шаблоне MVC, в отличие от Angular. Последний не содержит элементов для создания пользовательского интерфейса.

Ember.js — это фреймворк с открытым исходным кодом, основанный на MVC; служит для разработки веб-приложений. Содержит механизм маршрутизации и поддерживает двухстороннюю привязку данных. В коде этого фреймворка используется множество соглашений, что повышает продуктивность разработчиков ПО.

Jasmine — фреймворк с открытым исходным кодом, предназначенный для тестирования кода JavaScript. Не требует наличия объекта DOM. Содержит набор функций, проверяющих, ведут ли себя части приложения запланированным образом. Нередко используется вместе с Karma — программой для запуска тестов, которая позволяет проводить проверки в разных браузерах.

1.1.3. Библиотеки

Библиотеки, рассмотренные в этом подразделе, служат для разных целей и могут быть задействованы в веб-приложениях вместе с другими фреймворками или самостоятельно.

jQuery — популярная библиотека для JavaScript. Довольно проста в использовании и не требует значительного изменения стиля написания кода для веб-программ. Позволяет находить объекты DOM и манипулировать ими, а также обрабатывать события браузера и справляться с несовместимостью браузеров. Это расширяемая библиотека, разработчики со всего мира создали для нее тысячи плагинов. Если вы не можете найти плагин, который отвечает вашим нуждам, то всегда можете создать его самостоятельно.

Bootstrap — библиотека компонентов для создания пользовательского интерфейса с открытым исходным кодом, разработанная компанией Twitter. Они строятся согласно принципам адаптивного веб-дизайна, что значительно повышает ценность библиотеки, если ваше веб-приложение должно автоматически подстраивать свой макет в зависимости от размера экрана устройства пользователя. В этой книге мы будем использовать Bootstrap при разработке приложения-примера.

ПРИМЕЧАНИЕ

> В компании Google была разработана библиотека компонентов под названием Material Design, которая может стать альтернативой Bootstrap. Она оптимизирована для использования на разных устройствах и поставляется вместе с набором интересных элементов пользовательского интерфейса.

React — созданная компанией Facebook библиотека с открытым исходным кодом, предназначенная для сборки пользовательских интерфейсов. Представляет

собой слой V в аббревиатуре MVC. Не внедряется глубоко, и ее можно применять вместе с любой другой библиотекой или фреймворком. Создает собственный виртуальный объект DOM, минимизируя доступ к объекту DOM браузера, в результате чего повышается производительность. Что касается отрисовки содержимого, React вводит формат JSX — расширение синтаксиса JavaScript, которое выглядит как XML. Использование JSX рекомендуется, но не обязательно.

Polymer — библиотека, созданная компанией Google для сборки пользовательских компонентов на основе стандарта Web Components. Поставляется вместе с набором интересных настраиваемых элементов пользовательского интерфейса, которые можно включить в разметку HTML в виде тегов. Кроме того, содержит компоненты приложений, предназначенных для работы в режиме офлайн, а также элементы, использующие разнообразные API от Google (например, календарь, карты и др.).

RxJS — набор библиотек, необходимых для создания асинхронных программ и программ, основанных на событиях, с использованием наблюдаемых коллекций. Позволяет приложениям работать с асинхронными потоками данных наподобие серверного потока котировок акций или событий, связанных с движением мыши. С помощью RxJS потоки данных представляются в виде наблюдаемых последовательностей. Эту библиотеку можно применять как с другими фреймворками JavaScript, так и без них. В главах 5 и 8 вы увидите примеры использования наблюдаемых последовательностей в Angular.

Чтобы увидеть статистику, которая показывает, на каких сайтах задействованы те или иные фреймворки, вы можете посетить страницу BuiltWith: http://trends.builtwith.com/javascript.

Переход от Flex к Angular

Мы работаем на компанию Farata Systems, которая за много лет разработала много довольно сложного ПО, используя фреймворк Flex от Adobe. Это очень продуктивный фреймворк, созданный на основе строго типизированного скомпилированного языка ActionScript. Приложения, написанные с его помощью, развертываются в плагине для браузера Flash Player (применяется как виртуальная машина). Когда веб-сообщество начало отходить от плагинов, мы потратили два года, пытаясь найти замену Flex. Мы экспериментировали с разными фреймворками, основанными на JavaScript, но эффективность труда наших разработчиков серьезно падала. Наконец, мы увидели свет в конце туннеля, когда объединили язык TypeScript, фреймворк Angular и библиотеку для работы с пользовательским интерфейсом, такую как Angular Material.

1.1.4. Что такое Node.js

Node.js (или просто *Node*) — не просто фреймворк или библиотека. Это еще и среда времени выполнения. На протяжении большей части книги мы будем использовать время выполнения Node для запуска различных утилит наподобие Node Package Manager (npm). Например, для установки TypeScript можно запустить npm из командной строки:

```
npm install typescript
```

Фреймворк Node.js используется для разработки программ на языке JavaScript, которые функционируют вне браузера. Вы можете создать серверный слой веб-приложения на JavaScript или Typescript; в главе 8 вы напишете веб-сервер, применяя Node. Компания Google разработала для браузера Chrome высокопроизводительный движок JavaScript V8. Его можно задействовать для запуска кода, написанного с помощью API Node.js. Фреймворк Node.js включает в себя API для работы с файловой системой, доступа к базе данных, прослушивания запросов HTTP и др.

Члены сообщества JavaScript создали множество утилит, которые будут полезны при разработке веб-приложений. Используя движок JavaScript для Node, вы можете запускать их из командной строки.

1.2. Общий обзор AngularJS

Теперь вернемся к основной теме нашей книги: фреймворку Angular. Этот раздел — единственный, посвященный AngularJS.

Перечислим причины, по которым AngularJS стал таким популярным.

- Фреймворк содержит механизм создания пользовательских тегов и атрибутов HTML с помощью концепции директив, которая позволяет расширять набор тегов HTML в соответствии с потребностями приложения.
- AngularJS не слишком глубоко внедряется. Вы можете добавить атрибут `ng-app` к любому тегу `<div>`, и AngularJS будет управлять содержимым лишь этого тега. Остальная часть веб-страницы может быть написана на чистом HTML и JavaScript.
- Фреймворк позволяет легко связывать данные с представлениями. Изменение данных приводит к автоматическому обновлению соответствующего элемента представления, и наоборот.
- AngularJS поставляется с настраиваемым маршрутизатором; он разрешает соотносить шаблоны URL с соответствующими компонентами приложения, которые изменяют представление на веб-странице в зависимости от установленных соотношений.
- Поток данных приложения определяется в *контроллерах*, они являются объектами JavaScript, содержащими свойства и функции.
- Приложения, созданные с помощью AngularJS, используют иерархию *областей видимости* — объектов, необходимых для сохранения данных, общих для контроллеров и представлений.
- Фреймворк содержит модуль внедрения зависимостей, который позволяет разрабатывать слабо связанные приложения.

В то время как в jQuery были упрощены манипуляции с объектом DOM, AngularJS позволяет разработчикам отвязывать логику приложения от интерфейса путем структурирования приложения по шаблону проектирования MVC. На рис. 1.1 показан пример рабочего потока приложения, созданного с помощью этого фреймворка.

Рис. 1.1. Пример архитектуры приложения, написанного с использованием AngularJS

AngularJS может управлять всем веб-приложением. Для этого включите директиву `ng-app` в HTML-тег `<body>`:

```
<body ng-app="ProductApp">
```

На рис. 1.1, чтобы получить данные о продукте, пользователь загружает приложение ❶ и вводит идентификатор продукта ❷. Представление оповещает контроллер ❸, который обновляет модель ❹ и выполняет HTTP-запрос ❺ на удаленный сервер, используя сервис `$http`. AngularJS заполняет свойства модели на основе полученных данных ❺, и изменения в модели автоматически отразятся в пользовательском интерфейсе с помощью *связывающего выражения* ❻. После этого пользователь увидит данные о запрошенном продукте ❼.

AngularJS автоматически обновляет представление при модификации данных модели. Изменения в пользовательском интерфейсе вносятся в модель, если пользователь меняет данные в элементах управления представления, связанных с вводом. Такой двунаправленный механизм обновлений называется *двухсторонней привязкой*, он показан на рис. 1.2.

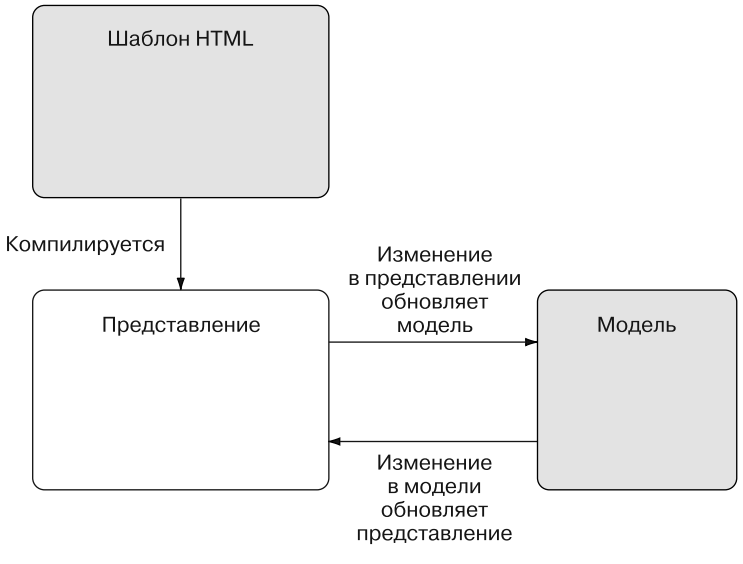


Рис. 1.2. Двухсторонняя привязка

В AngularJS модель и представление тесно связаны, поскольку двухсторонняя привязка означает, что один из элементов пары автоматически обновляет другой. Подобная возможность автоматического обновления очень удобна, но имеет свою цену.

При каждом обновлении модели фреймворк запускает специальный цикл `$digest`, который проходит по всему приложению, применяет привязку данных

и обновляет DOM, когда это необходимо. Каскадные обновления приводят к нескольким запускам цикла `$digest`, что может повлиять на производительность крупных приложений, использующих двухстороннюю привязку.

Манипуляции с объектом DOM браузера — самая медленная операция. Чем меньше приложение обновляет DOM, тем лучше оно работает.

Данные модели существуют в контексте определенного объекта `$scope`, области видимости AngularJS формируют иерархию объектов. Элемент `$rootScope` создается для целого приложения. Контроллеры и директивы (пользовательские компоненты) имеют собственные объекты `$scope`, и принципы работы областей видимости в фреймворке могут быть сложны для понимания.

Вы можете реализовать модульность, создавая и загружая объекты `module`. Когда конкретный модуль зависит от других объектов (таких как контроллеры, модули или сервисы), экземпляры этих объектов создаются и *внедряются* путем использования механизма внедрения зависимостей, представленного в AngularJS. В следующем фрагменте кода показывается один из способов, которым фреймворк внедряет один объект в другой:

```
var SearchController = function ($scope) {
        //..
};

SearchController['$inject'] = ['$scope'];

angular.module('auction').controller('SearchController', SearchController);
```

Определяет SearchController как функцию-конструктор, имеющую параметр $scope

Добавляет свойство $inject в контроллер, давая команду внедрить объект $scope в функцию-конструктор

Указывает, что объект SearchController должен стать контроллером модуля аукциона

В этом фрагменте кода квадратными скобками обозначен массив, и AngularJS может внедрять сразу несколько объектов следующим образом: `['$scope', 'myCustomService']`.

Данный фреймворк зачастую используется для создания одностраничных приложений, в которых лишь отдельные фрагменты страницы (подпредставления) обновляются в результате действий пользователя или из-за отправки данных на сервер. Хорошим примером таких подпредставлений является веб-приложение, показывающее котировки: при продаже акции обновляется лишь элемент, содержащий значение цены.

Навигация между представлениями в AngularJS выполняется с помощью конфигурирования компонента маршрутизатора `ng-route`. Можно указать количество параметров `.when`, чтобы направить приложение к соответствующему представлению в зависимости от шаблона URL. В следующем фрагменте кода

маршрутизатору предписывается использовать разметку из файла `home.html` и контроллер `HomeController`, если только URL не содержит элемент `/search`. В этом случае представление отрисует страницу `search.html`, и в качестве контроллера будет применен объект `SearchController`:

```
angular.module('auction', ['ngRoute'])
    .config(['$routeProvider', function ($routeProvider) {
        $routeProvider
          .when('/', {
             templateUrl: 'views/home.html',
             controller: 'HomeController' })
          .when('/search', {
              templateUrl: 'views/search.html',
              controller: 'SearchController' })
          .otherwise({
              redirectTo: '/'
    });
}]);
```

Маршрутизатор фреймворка поддерживает глубокое связывание, которое представляет собой способность поместить в закладки не только целую веб-страницу, но и определенное состояние внутри нее.

Теперь, после общего обзора AngularJS, взглянем, что нам может предложить Angular 2.

1.3. Общий обзор Angular

Фреймворк Angular гораздо производительнее, чем AngularJS. Он более понятен для освоения; архитектура приложений была упрощена, и код с его помощью легче читать и писать.

В этом разделе приводится краткий обзор Angular, в котором выделяются его более сильные стороны относительно AngularJS. Подробный архитектурный обзор Angular доступен в документации продукта на https://angular.io/guide/architecture.

1.3.1. Упрощение кода

Во-первых, Angular-приложение содержит стандартные модули в форматах ECMAScript 6 (ES6), Asynchronous Module Definition (AMD) и CommonJS. Обычно один модуль находится в одном файле. Нет необходимости прибегать к синтаксису, характерному для фреймворков, для загрузки и использования модулей. Запустите универсальный загрузчик модулей SystemJS (он рассматривается в главе 2) и добавьте операторы импорта, чтобы применить функциональные возможности, реализованные в загруженных модулях. Вам не нужно волноваться

насчет правильного использования тегов `<script>` в ваших файлах HTML. Если для модуля A требуются функции, содержащиеся в модуле B, просто импортируйте модуль B в модуль A.

Файл HTML для посадочной страницы вашего приложения содержит модули Angular и их зависимости. Код приложения загружается путем загрузки корневого модуля приложения. Все необходимые компоненты и сервисы будут загружены на основании объявлений в операторах `module` и `import`.

В следующем фрагменте кода показано типичное содержимое файла `index.html` приложения, созданного с помощью Angular, куда вы можете включить требуемые модули фреймворка. В сценарий `systemjs.config.js` входит конфигурация загрузчика SystemJS. Вызов `System.import('app')` загружает высокоуровневый элемент приложения, сконфигурированный в файле `systemjs.config.js` (показан в главе 2). Пользовательский тег `<app>` представляет собой значение, определенное в свойстве `selector` корневого компонента:

```
<!DOCTYPE html>
<html>
<head>
  <title>Angular seed project</title>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">

  <script src="node_modules/core-js/client/shim.min.js"></script>
  <script src="node_modules/zone.js/dist/zone.js"></script>
  <script src="node_modules/typescript/lib/typescript.js"></script>
  <script src="node_modules/systemjs/dist/system.src.js"></script>
  <script src="node_modules/rxjs/bundles/Rx.js"></script>
  <script src="systemjs.config.js"></script>
  <script>
    System.import('app').catch(function(err){ console.error(err); });
  </script>
</head>

<body>
<app>Loading...</app>
</body>
</html>
```

HTML-фрагмент в каждом приложении содержится либо внутри компонента (свойство `template`), либо в файле, на который ссылается этот компонент с помощью свойства `templateURL`. Второй вариант позволяет дизайнерам работать над интерфейсом вашего приложения, не изучая Angular.

Компонент Angular является центральным элементом новой архитектуры. На рис. 1.3 показана общая схема примера Angular-приложения, состоящего из четырех компонентов и двух сервисов; все они находятся внутри модуля внедрения зависимостей (Dependency Injection, DI). Этот модуль внедряет сервис `Http` в сервис `Service1`, а тот, в свою очередь, внедряется в компонент `GrandChild2`. Эта

схема отличается от приведенной рис. 1.1, где был показан принцип работы AngularJS.

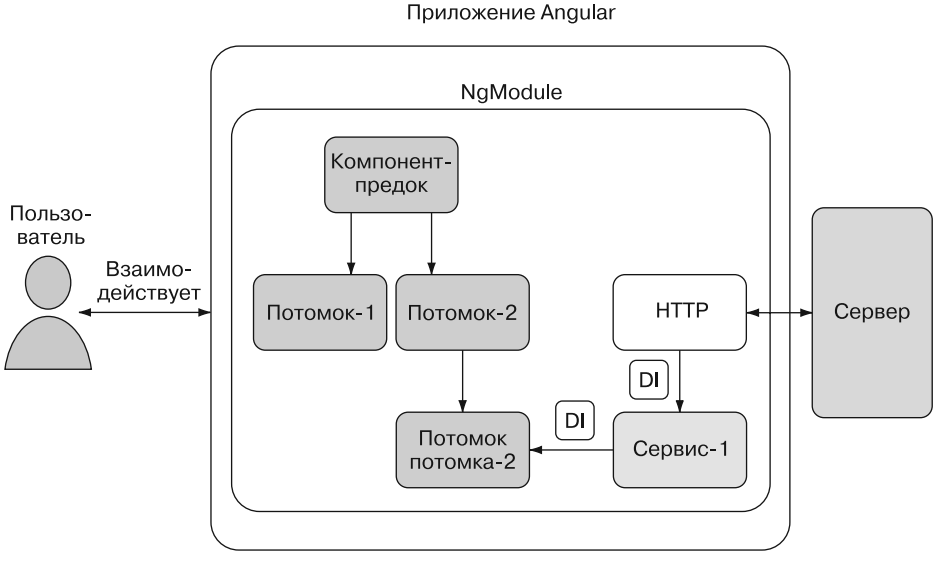


Рис. 1.3. Пример архитектуры Angular-приложения

Самый простой способ объявления компонента заключается в написании класса с помощью TypeScript (можно использовать ES5, ES6 или Dart). В приложении Б мы кратко расскажем о том, как писать компоненты Angular на TypeScript, а также приведем пример кода. Попробуйте понять этот код, прочитав минимальное количество пояснений.

Класс TypeScript содержит аннотацию метаданных `@NgModule` и представляет собой модуль. Класс TypeScript включает аннотацию метаданных `@Component` и представляет собой компонент. Аннотация `@Component` (также известная как декоратор) имеет свойство `template`, указывающее на фрагмент HTML, который должен быть отрисован в браузере.

Аннотации метаданных дают возможность изменять свойства компонентов во время разработки. Шаблон HTML может включать в себя выражения для привязки данных, окруженных двойными фигурными скобками. Ссылки на обработчики событий помещены в свойство `template` аннотации `@Component` и реализованы как методы класса. Еще одним примером аннотации метаданных выступает аннотация `@Injectable`, позволяющая отметить элемент, с которым должен работать модуль DI.

Аннотация `@Component` также содержит селектор, объявляющий имя пользовательского тега, который будет использован в документе HTML. Когда Angular видит элемент HTML, чье имя соответствует селектору, он знает, какой элемент

его реализует. В следующем фрагменте HTML показан родительский компонент `<auction-application>` с одним потомком, `<search-product>`:

```
<body>
  <auction-application>
    <search-product [productID]= "123"></search-product>
  </auction-application>
</body>
```

Предок отправляет данные потомкам путем привязки к входным свойствам потомка (обратите внимание на квадратные скобки в предыдущем фрагменте кода), а потомок общается с предками, отправляя события с помощью своих выходных свойств.

В конце главы вы найдете рис. 1.7, на котором показана главная страница (компонент-предок), чьи элементы-потомки окружены толстыми границами.

В следующем фрагменте кода показывается класс `SearchComponent`. Вы можете включить его в документ HTML как `<search-product>`, поскольку его объявление содержит свойство `selector` с таким же именем:

```
@Component({
  selector: 'search-product',
  template:
    `<form>
      <div>
        <input id="prodToFind" #prod>
        <button (click)="findProduct(prod.value)">Find Product</button>
        Product name: {{product.name}}
      </div>
    `</form>

})
  class SearchComponent {
    @Input() productID: number;

    product: Product; // Опустим код класса Product

    findProduct(prodName: string){
    // Здесь будет расположен реализация обработчика щелчков кнопкой мыши
  }
  // Здесь будет расположен другой код
}
```

Если вы знакомы с любым объектно-ориентированным языком, имеющим классы, то должны понять большую часть предыдущего фрагмента кода. Аннотированный класс `SearchComponent` объявляет переменную `product`, которая может представлять объект с несколькими свойствами, и одно из них (`name`) привязано к представлению (`{{product.name}}`). Локальная переменная шаблона `#prod` будет иметь ссылку на элемент `<input>`, поэтому вам не нужно запрашивать DOM для того, чтобы получить введенное значение.

Выражение `(click)` представляет собой событие щелчка кнопкой мыши. Функция обработчика событий получает значение аргумента из входного параметра `productID`, который будет заполнен родительским элементом путем привязки.

Мы кратко рассмотрели пример компонента. Более подробное описание содержимого этих элементов будет представлено в начале следующей главы. Не стоит беспокоиться, если вы раньше никогда не работали с классами, — данная тема раскрывается в приложениях А и Б. На рис. 1.4 показана внутренняя работа примера компонента, выполняющего поиск некоторых продуктов.

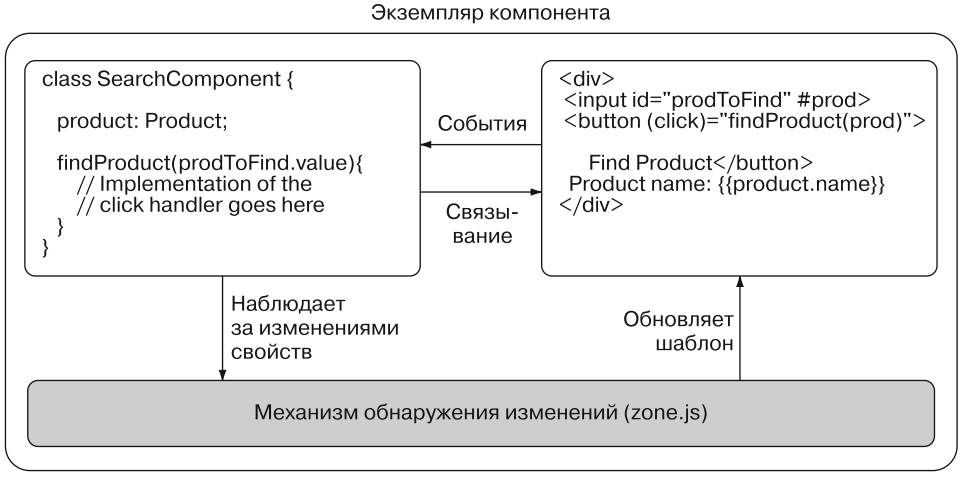


Рис. 1.4. Содержимое компонента

Компонент отрисовывает данные продукта, полученные из сервиса, представленного классом. В TypeScript класс `Product` будет выглядеть следующим образом:

```
class Product{
    id: number,
    name: string;
    description: string;
    bid: number;
    price: number;

    // Здесь будут расположены конструктор и другие методы
}
```

Обратите внимание: TypeScript позволяет объявить переменные класса с помощью типов. Чтобы указать компоненту пользовательского интерфейса `SearchComponent` на данные, можно объявить переменную класса, такую как `product`:

```
@Component({ /* код опущен для краткости */ })
class SearchComponent {
```

```
    product: Product;

    findProduct(productID){
        // Реализация обработчика щелчков кнопкой мыши
        // для раздела «Поиск компонентов» будет находиться здесь
    }
}
```

Если компонент поиска может вернуть несколько продуктов, то для их сохранения можно объявить массив:

```
products: Array<Product>;
```

Выражение *обобщений* объясняется в приложении Б. В предыдущем фрагменте кода конструкция `<Product>` указывает компилятору TypeScript, что в этом массиве можно хранить только объекты типа `Product`.

Фреймворк Angular не основан на шаблоне проектирования MVC, и ваше приложение не будет иметь отдельные контроллеры (C в аббревиатуре MVC). Компонент и внедренные сервисы (если таковые нужны) содержат весь необходимый код. В нашем примере класс `SearchProduct` будет включать код, который выполняет обязанности контроллера в дополнение к коду, необходимому для элемента пользовательского интерфейса представления HTML. В целях более четкого разделения TypeScript и HTML содержимое раздела `template` аннотации `@Component` можно хранить в отдельном файле, используя `templateUrl` вместо `template`, но это уже дело вкуса.

Теперь рассмотрим, почему дизайн Angular проще, чем в AngularJS. В последнем все директивы загружались в глобальную область видимости, а в Angular вы сами указываете необходимые директивы на уровне модулей, что предоставляет более качественную инкапсуляцию.

Вам не нужно работать с иерархией объектов областей видимости, как было в AngularJS. Фреймворк Angular основан на компонентах, и свойства создаются для этого объекта, который становится областью видимости компонента.

Одним из способов создания экземпляров объектов является применение оператора `new`. Если объект A зависит от объекта B, то в коде объекта A можно написать `let myB = new B();`. «Внедрение зависимостей» — шаблон проектирования, изменяющий способ создания объектов, который будет использован для кода. Вместо того чтобы явно создавать экземпляры объектов (например, с помощью оператора `new`), фреймворк будет создавать и внедрять их в код. Angular поставляется с модулем DI (эту тему мы рассмотрим в главе 4).

В AngularJS существует несколько способов регистрации зависимостей, что иногда может показаться непонятным. В Angular внедрять зависимости в элемент можно только с помощью конструктора. В следующем фрагменте кода TypeScript показывается, как бы вы внедряли компонент `ProductService` в компонент `SearchComponent`. Нужно лишь указать поставщик и объявить аргумент конструктора, используя тип, который соответствует типу поставщика:

```
@Component({
  selector: 'search-product',
  providers: [ProductService],
```

```
  template:`<div>...<div>`
})
class SearchComponent {
  products: Array<Product> = [];

  constructor(productService: ProductService) {
    this.products = productService.getProducts();
  }
}
```

Этот код не использует оператор `new` — Angular создаст объект типа `ProductService` и передаст ссылку на него компоненту `SearchComponent`.

Таким образом, Angular проще AngularJS по нескольким причинам:

- каждый блок вашего приложения является элементом, имеющим хорошо инкапсулированную функциональность представления, контроллера и автоматически сгенерированного детектора изменений;
- компоненты могут быть запрограммированы как аннотированные классы;
- вам не нужно работать с иерархией областей видимости;
- зависимые элементы внедряются в конструктор компонента;
- двусторонняя привязка по умолчанию выключена;
- механизм определения изменений был переписан и работает быстрее.

Концепции Angular легко поймут программисты, работающие с Java, C# и C++, — большая часть разработчиков промышленных программ. Нравится вам это или нет, но фреймворк становится более популярным, когда его начинают использовать для промышленных целей. AngularJS широко применяется в промышленности, и навыки работы с ним имеют спрос. Поскольку писать приложения на Angular проще, чем на AngularJS, данная тенденция будет сохраняться.

1.3.2. Улучшение производительности

Сайт Repaint Rate Challenge (http://mathieuancelin.github.io/js-repaint-perfs) сравнивает производительность отрисовки для разных фреймворков. Вы можете сравнить эффективность AngularJS и Angular 2 — последний демонстрирует значительное улучшение. Оно появилось в основном благодаря заново спроектированному внутреннему устройству фреймворка Angular. Отрисовка пользовательского интерфейса и API приложений была разделена на два слоя, что позволяет запускать не связанный с пользовательским интерфейсом код в отдельном рабочем потоке. В дополнение к возможности запускать код данных слоев параллельно браузеры выделяют разные ядра для этих потоков везде, где возможно. Вы можете найти детальное описание новой архитектуры отрисовки в документе, хранящемся в Google Docs, который называется *Angular 2 Rendering Architecture* — он доступен по ссылке https://docs.google.com/document/d/1M9FmT05Q6qpsjgvH1XvCm840yn2eWEg0PMskSQz7k4E/edit.

Создание отдельного слоя для отрисовки имеет еще одно важное преимущество: использование разных отрисовщиков для различных устройств. Каждый элемент содержит аннотацию `@Component`, которая включает шаблон HTML, определяющий его внешний вид. Если вы хотите создать компонент `<stock-price>` для отображения котировок акций в браузере, то его часть, что отвечает за пользовательский интерфейс, может выглядеть так:

```
@Component({
  selector: 'stock-price',
  template: '<div>The price of an IBM share is $165.50</div>'
})
class StockPriceComponent {
...
}
```

Графический движок Angular — отдельный модуль, позволяющий сторонним поставщикам заменить стандартный отрисовщик DOM на тот, что работает для платформ, не являющихся браузерами. Например, это дает возможность повторно использовать код TypeScript для устройств, имеющих сторонние отрисовщики пользовательского интерфейса, которые отрисовывают нативные компоненты. Часть их кода, написанная с помощью TypeScript, остается такой же, но содержимое свойства `template` декоратора `@Component` может содержать XML или другой язык для отрисовки нативных компонентов.

Один подобный отрисовщик для Angular уже реализован во фреймворке NativeScript, который служит мостом между JavaScript и нативными компонентами пользовательского интерфейса для iOS и Android. С помощью NativeScript можно повторно использовать код компонента, просто заменив HTML в шаблоне на XML. Еще один отрисовщик пользовательского интерфейса позволяет задействовать Angular вместе с React Native, что является альтернативным способом создания нативных (не гибридных) пользовательских интерфейсов для iOS и Android.

Новый, улучшенный механизм определения изменений также вносит свою лепту в улучшение производительности Angular. Этот фреймворк не использует двухстороннюю привязку, если только вы не укажете ее вручную. Односторонняя привязка упрощает определение изменений в приложении, которое может иметь множество взаимозависимых привязок. Вы можете пометить компонент, и он будет исключен из рабочего потока определения изменений, поэтому не будет проверяться при обнаружении изменения в другом элементе.

ПРИМЕЧАНИЕ

Несмотря на то что Angular является полной переработкой AngularJS, если вы применяете последний, то можете начать писать код в стиле Angular путем использования ng-forward (см. https://github.com/ngUpgraders/ng-forward). Другой подход (ng-upgrade) заключается в том, чтобы постепенно переходить на новую версию фреймворка, запуская Angular и AngularJS в одном приложении (см. https://angular.io/docs/ts/latest/guide/upgrade.html), но это увеличит его размер.

1.4. Инструментарий Angular-разработчика

Предположим, вам нужно нанять веб-разработчика, работавшего с Angular. Что, по вашему мнению, должен знать такой специалист? Он должен понимать архитектуру, компоненты и концепции приложений Angular, упомянутые в предыдущих разделах, но этого недостаточно. В следующем относительно длинном списке приводятся языки и инструменты, которыми пользуются профессиональные Angular-разработчики. Не все они нужны для разработки и развертывания конкретных приложений. В рамках данной книги мы будем применять только половину из них.

- JavaScript де-факто является стандартным языком программирования для создания клиентской части веб-приложений.
- TypeScript — надмножество JavaScript, позволяющее повысить продуктивность разработчиков. TypeScript поддерживает большую часть функциональности ES6 и добавляет необязательные типы, интерфейсы, аннотации метаданных и пр.
- Анализатор кода TypeScript использует файлы с определением типов для кода, который изначально не был написан на TypeScript. DefinitelyTyped представляет собой популярную коллекцию подобных файлов, содержащих описание API сотен библиотек и фреймворков JavaScript. Определение типов позволяет IDE предоставлять помощь в зависимости от контекста и выделять ошибки. Вы будете устанавливать файлы с определением типов с помощью структуры `@types` (см. приложение Б).
- Поскольку большинство браузеров поддерживают лишь синтаксис ECMAScript 5 (ES5), для развертывания вам понадобится *скомпилировать* (преобразовать из одного языка в другой) код, написанный на TypeScript или ES6, к ES5. Angular-разработчики могут использовать для этих целей Babel, Traceur и компилятор TypeScript (для получения подробной информации см. приложения А и Б).
- SystemJS — универсальный загрузчик модулей, который загружает модули, созданные в соответствии со стандартами ES6, AMD и CommonJS.
- Angular CLI — генератор кода, позволяющий генерировать новые проекты, компоненты, сервисы и маршруты для Angular, а также создавать приложение для развертывания.
- Node.js — платформа, построенная на движке JavaScript для Chrome. Содержит фреймворк и среду выполнения для запуска кода JavaScript за пределами браузера. В рамках данной книги вы не будете использовать этот фреймворк, однако среда выполнения понадобится для установки инструментов, необходимых при разработке приложений Angular.
- npm — менеджер пакетов, позволяющий загружать инструменты, а также библиотеки и фреймворки JavaScript. Этот менеджер пакетов имеет репозиторий, в котором содержатся тысячи элементов. Вы будете использовать его для

установки практически всего: от инструментов разработчика (например, компилятора TypeScript) до зависимостей приложений (таких как Angular, jQuery и др.). С помощью npm можно запускать сценарии. Вы будете использовать эту функциональность, чтобы запускать серверы HTTP, а также для автоматизации сборки.

- Bower раньше был популярным менеджером пакетов, предназначенным для разрешения зависимостей приложений (наподобие Angular и jQuery). Мы больше не будем использовать его, поскольку все, что нужно, можно загрузить с помощью npm.
- jspm — еще один менеджер пакетов. Зачем он нужен, если npm может позаботиться обо всех зависимостях? Современные веб-приложения состоят из загружаемых модулей, и jspm интегрирует SystemJS, что позволяет без труда загружать подобные модули. В главе 2 мы кратко сравним npm и jspm.
- Grunt — средство для запуска задач. Между разработкой и развертыванием находится много этапов, и все они должны быть автоматизированы. Вам может понадобиться скомпилировать код, написанный на TypeScript или ES6, в более широко поддерживаемый синтаксис ES5, а код, изображения и файлы CSS — минимизировать. Кроме того, может воникнуть необходимость включить все задачи, которые проверяют качество кода, и модульные тесты для вашего приложения. Grunt поможет сконфигурировать все задачи и их зависимости, используя файл JSON, поэтому процесс будет на 100 % автоматизирован.
- Gulp — еще одно средство для запуска задач. Оно может автоматизировать задачи точно так же, как и Grunt, но конфигурировать вы будете с помощью не JSON, а JavaScript. Это позволит при необходимости выполнить отладку.
- JSLint и ESLint — анализаторы кода, которые определяют проблемные шаблоны в программах JavaScript или документах, отформатированных в JSON. Они являются инструментами проверки качества кода. Запуск программы JavaScript с помощью JSLint или ESLint приведет к появлению предупреждений, указывающих на способы улучшения качества кода программы.
- TSLint — инструмент проверки качества кода для TypeScript. Он имеет набор расширяемых правил для навязывания рекомендованного стиля написания кода и шаблонов.
- Minifiers, как и UglifyJS, уменьшает размер файлов. В JavaScript эти программные средства удаляют комментарии и line breaks, а также укорачивают имена переменных. Минификацию можно выполнить для HTML, CSS и файлов изображений.
- Упаковщики, такие как Webpack, объединяют несколько файлов и их зависимости в один файл.
- Поскольку синтаксис JavaScript очень либерален, для кода приложения требуется тестирование, поэтому нужно выбрать один из фреймворков тестирования. В данной книге вы будете использовать фреймворк Jasmine и средство для запуска тестов Karma.

- JavaScript и TypeScript широко поддерживаются современными IDE и текстовыми редакторами, такими как WebStorm, Visual Studio, Visual Studio Code, Sublime Text, Atom и др.
- Все крупные браузеры поставляются с инструментами разработчика, которые позволяют выполнять отладку программ прямо в браузере. Даже если программа была написана на TypeScript и развернута в JavaScript, вы можете отладить оригинальный исходный код, используя его отображения. Мы применяем Chrome Developer Tools.
- Веб-приложения должны работать на мобильных устройствах. Вам следует задействовать компоненты пользовательского интерфейса, которые поддерживают адаптивный подход к веб-дизайну с целью гарантировать, что макет пользовательского интерфейса изменяется в зависимости от размера экрана устройства пользователя[1].

Представленный список может выглядеть устрашающе, но вам не нужно использовать каждый из его компонентов. В нашей книге вы будете применять следующие инструменты:

- npm для конфигурирования приложений, установки утилит и зависимостей. Вы будете задействовать сценарии npm, чтобы запускать веб-серверы и задачи при автоматизации сборки;
- Node.js в качестве среды выполнения для запускаемых утилит, а также как фреймворк для написания веб-сервера (глава 8);
- SystemJS для загрузки кода приложения и динамического компилирования TypeScript в браузере;
- компилятор командной строки TypeScript под названием tsc для запуска примеров из приложения Б и программирования приложения с помощью Node в главе 8;
- Jasmine для создания модульных тестов и Karma для их запуска (глава 9);
- Webpack для минимизации и упаковки приложений для развертывания (глава 10).

ПРИМЕЧАНИЕ

Писать программы на Angular гораздо проще, чем на AngularJS. Однако вам придется серьезно подготовиться, поскольку вы будете использовать средства компиляции в другие языки и загрузчики модулей, которые не нужны при разработке приложения с помощью JavaScript и AngularJS. Как правило, введение модулей ES6 изменяет способ загрузки приложений в браузер в будущем, и мы будем применять этот новый подход в рамках книги.

[1] Чтобы узнать больше об адаптивном веб-дизайне, обратитесь к «Википедии»: https://en.wikipedia.org/wiki/Responsive_web_design.

На рис. 1.5 показано, как инструменты могут применяться на разных этапах процессов разработки и развертывания. Инструменты, которые вы будете задействовать в данной книге, выделены полужирным шрифтом.

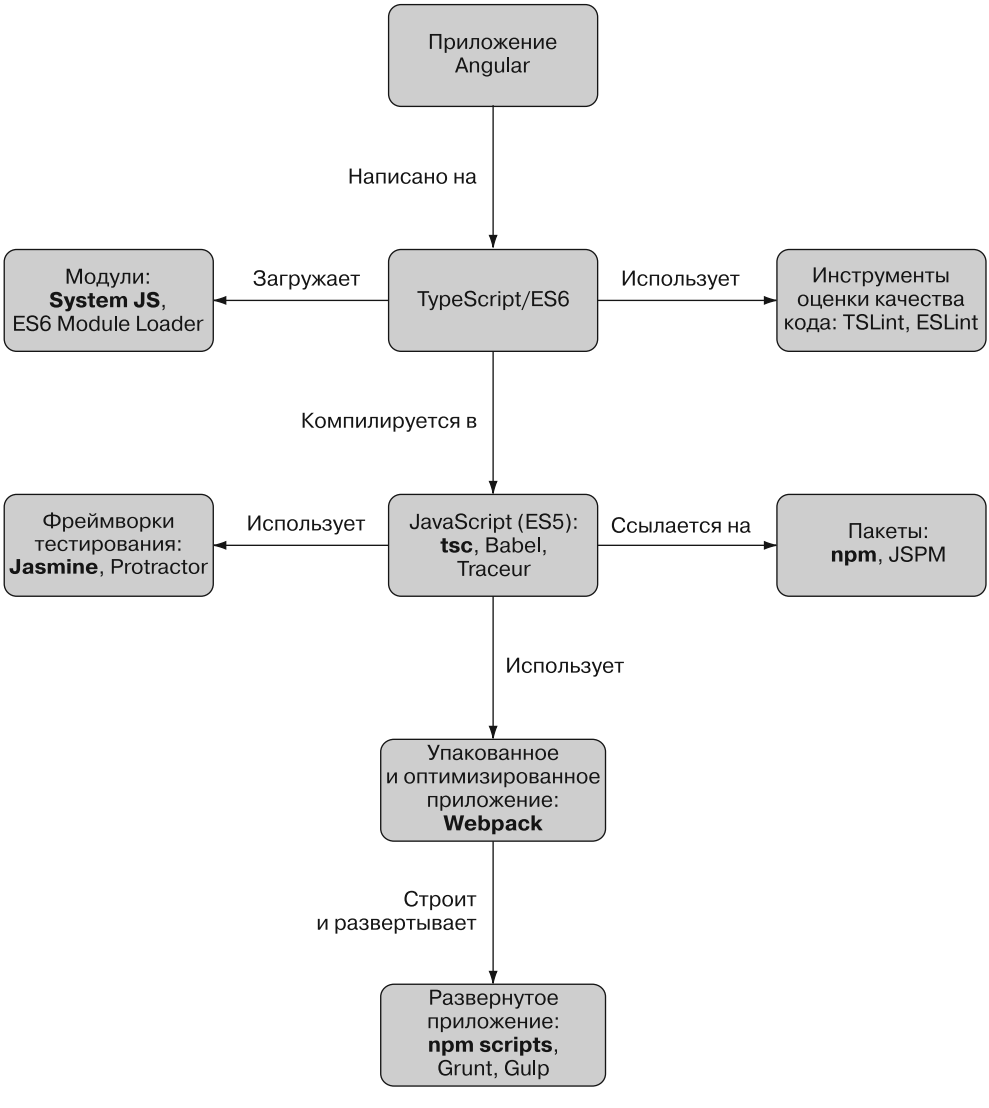


Рис. 1.5. Используемые инструменты

Программировать на Angular проще, чем на AngularJS, но настраивать среду нужно правильно, чтобы вы могли насладиться процессом разработки. В следующей главе мы обсудим настройку проектов и инструментов более подробно.

1.5. Как все делается в Angular

Чтобы вы получили представление о том, как все делается в Angular, мы составили табл. 1.1. В ней перечисляются задачи, которые вам, возможно, понадобится решить (левый столбец), а также способы решения этих задач с помощью комбинации из Angular/TypeScript (правый столбец). Здесь приведен неполный список задач, и мы покажем лишь фрагменты синтаксиса, чтобы вы получили общее представление. Подробнее вся функциональность будет описана в книге.

Таблица 1.1. Как все делается в Angular

Задача	**Способ решения**
Реализация бизнес-логики	Создайте класс, и Angular создаст его объект и внедрит в компонент. Вы можете также использовать оператор new
Реализация компонента с пользовательским интерфейсом	Создайте класс с аннотацией @Component
Определение шаблона HTML для отрисовки компонентом	Укажите либо код HTML в аннотации @Component с помощью свойства template, либо имя файла HTML в templateURL
Манипуляции с HTML	Примените одну из структурных директив (*ngIf, *ngFor) или создайте собственный класс с аннотацией @Directive
Отсылка к переменной класса текущего объекта	Задействуйте ключевое слово this: this.userName="Mary";
Настройка навигации для приложения с одной страницей	Сконфигурируйте основанный на компонентах маршрутизатор, позволяющий соотносить компоненты и сегменты URL, и добавьте тег <router-outlet> к шаблону там, где вы хотите отрисовать элемент
Отображение значения свойства компонента пользовательского интерфейса	Разместите переменные внутри двойных фигурных скобок внутри шаблона: {{customerName}}
Привязка свойства компонента и пользовательского интерфейса	Используйте привязку свойств и квадратные скобки: <input [value]="greeting" >
Обработка событий пользовательского интерфейса	Окружите имя события круглыми скобками и укажите обработчик: <button (click)="onClickEvent()">Get Products</button>
Использование двухсторонней привязки	Задействуйте нотацию [()]:<input [(ngModel)] = "myComponentProperty">
Передача данных компоненту	Укажите для компонентов аннотации @Input и привяжите к ним значения
Передача данных из компонента	Укажите для компонентов аннотации @Output и используйте EventEmitter для отправки событий
Создание запроса HTTP	Внедрите объект HTTP в компонент и вызовите один из методов HTTP: this.http.get('/products')

Задача	Способ решения
Обработка ответов HTTP	Примените метод subscribe() для результата, который поступает в формате наблюдаемого потока: this.http.get('/products').subscribe(...);
Передача фрагмента HTML компоненту-потомку	Используйте тег <ng-content> в шаблоне потомка
Перехватывание изменения компонентов	Задействуйте привязки для жизненного цикла элемента
Развертывание	Используйте сторонние упаковщики наподобие Webpack для упаковки файлов приложений и фреймворков в пакеты JavaScript

1.6. Знакомство с приложением-примером

Чтобы сделать книгу более практичной, мы будем начинать каждую главу с демонстрации небольших приложений, которые иллюстрируют синтаксис или приемы работы с Angular. В конце каждой главы вы используете эти новые концепции и увидите, как компоненты и сервисы объединяются в рабочее приложение.

Представьте онлайн-аукцион, где люди могут просматривать и искать продукты. После отображения результата пользователь может выбрать продукт и сделать на него ставку. Каждая новая ставка будет проверяться на сервере, в результате чего будет либо принята, либо отклонена. Информация о последних ставках будет отправлена сервером всем пользователям, подписанным на подобные уведомления.

Функции просмотра, поиска и размещения ставок будут воплощены путем запросов к конечным RESTful-точкам, реализованным на сервере, который создан с помощью Node.js. Сервер будет использовать WebSockets для отправки уведомлений о принятии или отклонении ставки, а также о ставках, сделанных другими пользователями. На рис. 1.6 показан пример рабочего потока онлайн-аукциона.

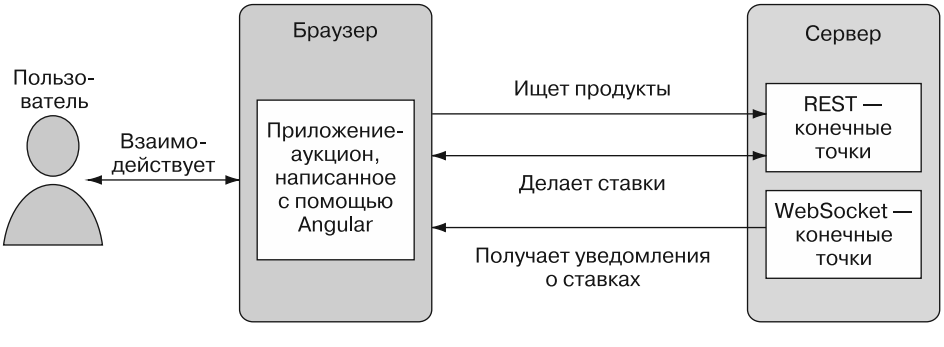


Рис. 1.6. Рабочий процесс онлайн-аукциона

На рис. 1.7 показано, как главная страница аукциона будет отрисована на персональных компьютерах. Поначалу мы будем использовать серые заполнители вместо изображений продуктов.

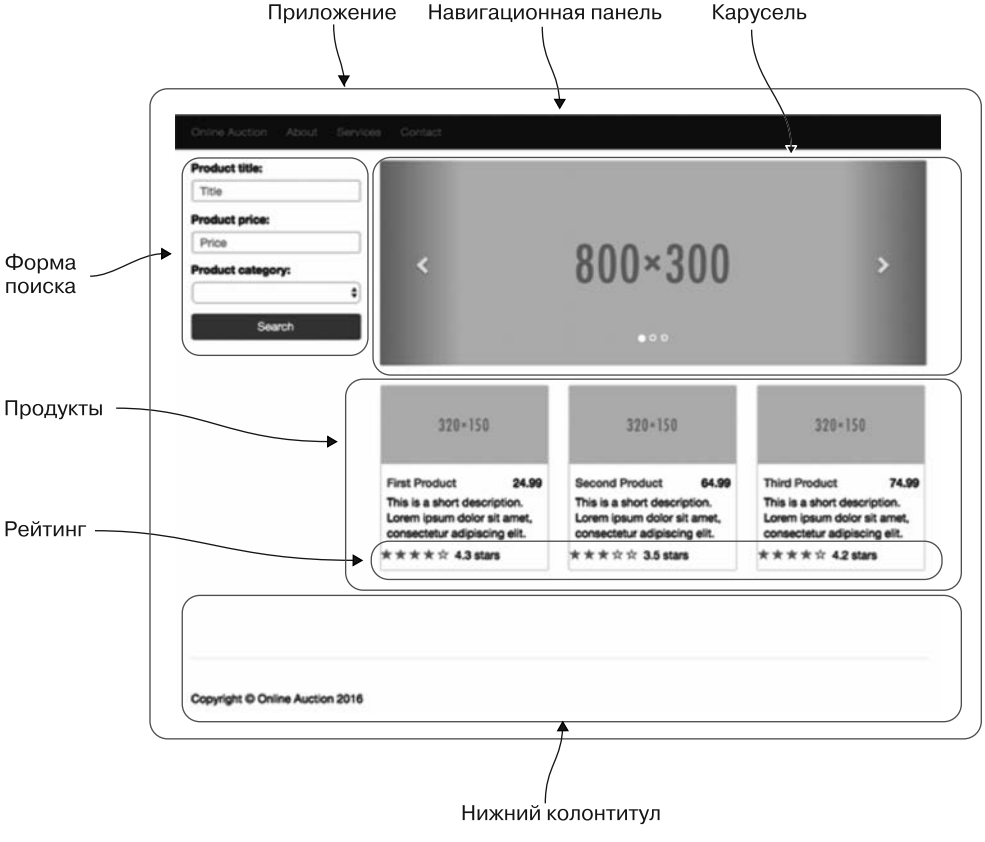


Рис. 1.7. Главная страница онлайн-аукциона, на которой выделены компоненты

Вы будете применять адаптивные элементы пользовательского интерфейса, поэтому на смартфонах главная страница будет отрисована так, как показано на рис. 1.8.

Разработка приложения на Angular сводится к созданию и сборке компонентов. Код онлайн-аукциона будет написан с помощью TypeScript, а представление элементов — разработано как шаблоны HTML, имеющие привязку данных. На рис. 1.9 показана исходная структура проекта онлайн-аукциона.

Файл `index.html` лишь загрузит основной компонент приложения, представленный двумя файлами: `application.html` и `application.ts`. Он будет содержать другие элементы, такие как продукты, поиск и т. д.

Все зависимости компонента приложения будут загружены автоматически.

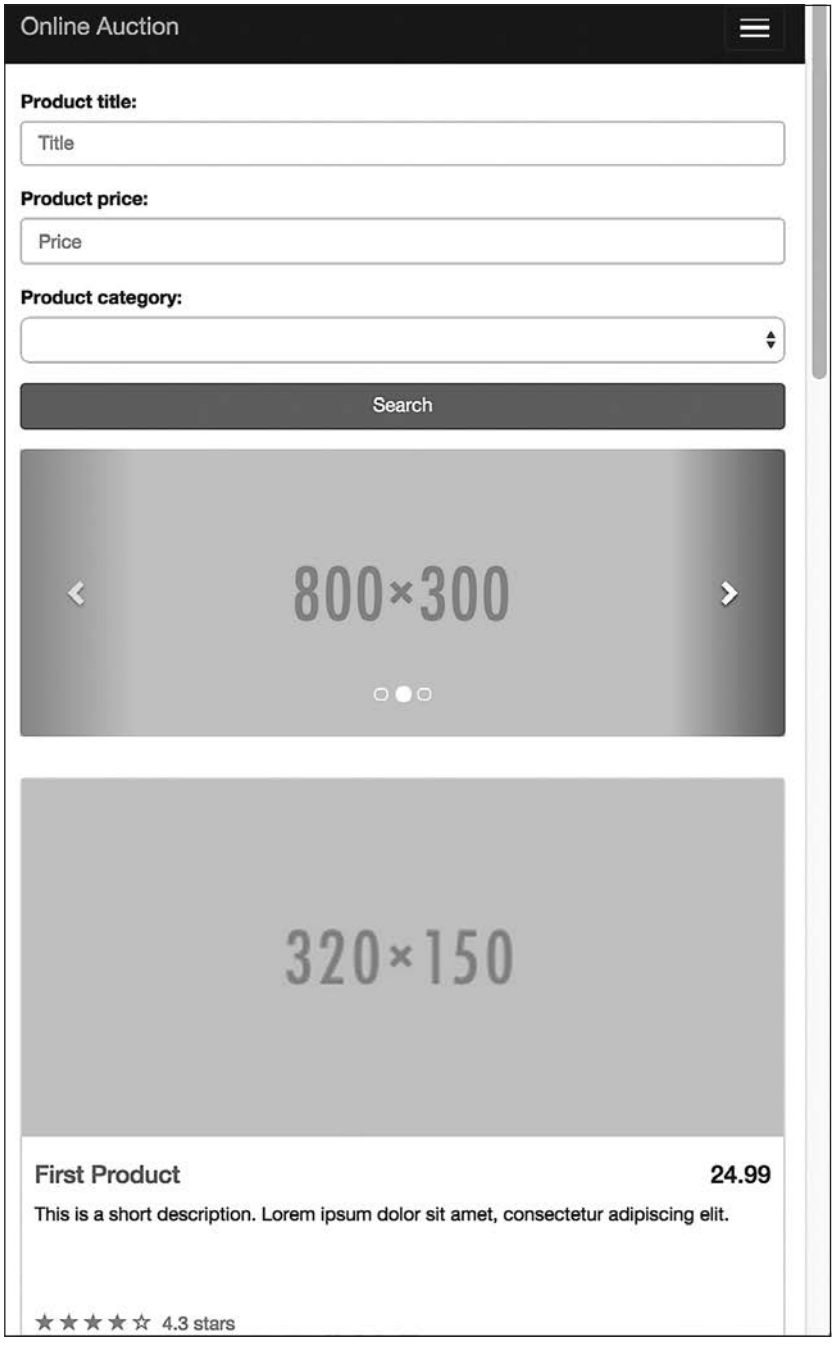


Рис. 1.8. Так выглядит на смартфоне главная страница аукциона

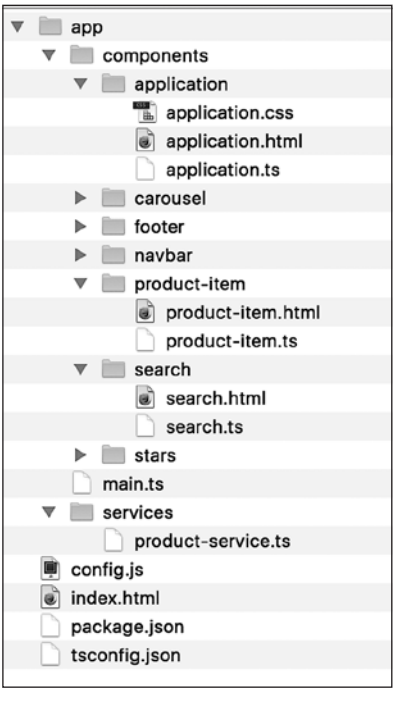


Рис. 1.9. Исходная структура проекта приложения-аукциона

1.7. Резюме

В данной главе мы бегло рассмотрели фреймворк Angular 2 и сравнили его с предыдущей версией, AngularJS. Мы также познакомились с приложением-примером, представляющим собой онлайн-аукцион, которое будем разрабатывать в процессе чтения этой книги.

- Архитектура Angular проще архитектуры AngularJS.
- Angular-приложения могут быть разработаны с помощью TypeScript или JavaScript.
- Исходный код должен быть скомпилирован в JavaScript перед развертыванием.
- Angular-разработчик должен быть знаком со многими инструментами.
- Angular — фреймворк, основанный на компонентах.
- Хорошие фреймворки позволяют разработчикам приложений писать меньше кода, и Angular — именно такой.

2 Приступаем к работе с Angular

В этой главе:

- написание вашего первого Angular-приложения;
- знакомство с универсальным загрузчиком модулей SystemJS;
- роль менеджеров пакетов;
- первая версия приложения для онлайн-аукциона.

В данной главе мы рассмотрим вопрос разработки Angular-приложений с помощью современных инструментов и веб-технологий, таких как аннотации, модули ES6 и загрузчики модулей. Angular изменяет привычный способ разработки приложений, написанных на JavaScript. Вы напишете три версии приложения Hello World. Кроме того, мы кратко рассмотрим менеджеры пакетов и универсальные загрузчики модулей SystemJS.

После этого мы создадим небольшой проект, который послужит шаблоном для создания ваших собственных проектов с помощью Angular. Далее мы рассмотрим основные составные части Angular-приложений, например компоненты и представления, а также внедрение зависимостей и привязку данных. В конце главы познакомимся с приложением для онлайн-аукционов, которое вы будете разрабатывать на протяжении книги.

СОВЕТ

Если вы не знакомы с синтаксисом TypeScript и ECMAScript 6, то рекомендуем обратиться к приложениям А и Б, а затем продолжать чтение с данной главы.

2.1. Первое приложение для Angular

В этом разделе мы покажем три версии приложения Hello World, разработанные с помощью TypeScript, ES5 и ES6. Только здесь вы увидите Angular-приложения,

написанные на ES5 и ES6, — все остальные фрагменты кода будут созданы с применением TypeScript.

2.1.1. Hello World с использованием TypeScript

Первое приложение выглядит довольно минималистично и поможет быстро начать работу с Angular. Оно состоит из двух файлов:

```
├── index.html
└── main.ts
```

Оба файла располагаются в каталоге `hello-world-ts` загружаемого кода для данной книги. Файл `index.html` — точка входа в приложение. Он содержит ссылки на фреймворк Angular, его зависимости, а также файл `main.ts`, включающий код для инициализации вашего приложения. Некоторые из этих ссылок могут находиться в конфигурационном файле загрузчика модулей (в рамках книги вы будете использовать загрузчики SystemJS и Webpack).

Загрузка ANGULAR в файле HTML

Код фреймворка Angular состоит из модулей (по одному файлу на модуль), которые объединяются в библиотеки, группируемые логически в пакеты наподобие `@angular/core`, `@angular/common` и пр. Ваше приложение должно загрузить требуемые пакеты до того, как будет загружен код приложения.

Теперь создадим файл `index.html`, работа которого будет начинаться с загрузки требуемых сценариев Angular, компилятора TypeScript и загрузчика модулей SystemJS.

В следующем фрагменте кода, приведенном в листинге 2.1, эти сценарии загружаются из сети распространения контента (content delivery network, CDN) https://unpkg.com/#/.

Листинг 2.1. Фрагмент содержимого файла TypeScript index.html

```
<!DOCTYPE html>
<html>
<head>
  <script src="//unpkg.com/zone.js@0.6.12"></script>
  <script src="//unpkg.com/typescript@2.0.0"></script>
  <script src="//unpkg.com/systemjs@0.19.37/dist/system.src.js"></script>
  <script src="//unpkg.com/core-js/client/shim.min.js"></script>
  <script>

    System.config({
```

Библиотека SystemJS динамически загружает код приложения в браузер

Zone.js — библиотека, запускающая механизм определения изменений

Компилятор TypeScript преобразует ваш код в код на JavaScript прямо в браузере

Конфигурирует загрузчик SystemJS так, чтобы он загружал и компилировал код TypeScript

```
      transpiler: 'typescript',
      typescriptOptions: {emitDecoratorMetadata: true},
      map: {
        'rxjs': 'https://unpkg.com/rxjs@5.0.0-beta.12',
        '@angular/core'                    : 'https://unpkg.com/@angular/
        ➥ core@2.0.0',
        '@angular/common'                  : 'https://unpkg.com/@angular/
        ➥ common@2.0.0',
        '@angular/compiler'                : 'https://unpkg.com/@angular/
        ➥ compiler@2.0.0',
        '@angular/platform-browser'        : 'https://unpkg.com/@angular/
        ➥ platform-browser@2.0.0',
        '@angular/platform-browser-dynamic': 'https://unpkg.com/@angular/
        ➥ platform-browser-dynamic@2.0.0'
      },

      packages: {
        '@angular/core'                    : {main: 'index.js'},
        '@angular/common'                  : {main: 'index.js'},
        '@angular/compiler'                : {main: 'index.js'},
        '@angular/platform-browser'        : {main: 'index.js'},
        '@angular/platform-browser-dynamic': {main: 'index.js'}
      }
    });
    System.import('main.ts');
  </script>
</head>
<body>
  <hello-world></hello-world>
</body>
</html>
```

Соотносит имена модулей Angular с их локациями CDN

Указывает основной сценарий для каждого модуля Angular

Дает SystemJS команду загрузить основной модуль из файла main.ts

Пользовательский HTM-элемент <hello-world></hello-world> представляет компонент, который реализован в файле main.ts

Когда приложение запущено, тег `<hello-world>` будет заменен содержимым шаблона из аннотации `@Component`, показанной в листинге 2.2.

СОВЕТ

Если вы используете Internet Explorer, то вам может понадобиться добавить дополнительный сценарий system-polyfills.js.

Сети распространения контента (Content delivery networks, CDNs)

unpkg (https://unpkg.com/#/) — сеть распространения контента для пакетов, опубликованных в реестре менеджера пакетов npm (https://www.npmjs.com/). Обратитесь к npmjs.com, чтобы найти последнюю версию необходимого вам пакета. Если хотите увидеть, какие еще версии этого пакета доступны, то запустите команду `npm info packagename`.

Сгенерированные файлы не отправляются в систему контроля версий, и Angular 2 не предоставляет готовых к использованию пакетов в своем репозитории Git. Они генерируются автоматически и публикуются вместе с пакетом npm (https://www.npmjs.com/~angular), в связи с чем вы можете применять unpkg так, чтобы непосредственно сослаться на готовые к выпуску пакеты из файлов HTML. Мы же предпочитаем задействовать версию Angular, установленную локально, а также ее зависимости, поэтому вы установите их с помощью npm в подразделе 2.4.2. Все, что было установлено с использованием npm, будет храниться в каталоге `node_modules` каждого проекта.

Файл TypeScript

Теперь создадим файл `main.ts`, который содержит код TypeScript/Angular и имеет три части.

1. Объявление компонента Hello World.
2. Оборачивание данного компонента в модуль.
3. Загрузка модуля.

Далее в главе вы реализуете приведенные части в отдельных файлах, но здесь для простоты мы разместим весь код этого небольшого приложения в одном файле (листинг 2.2).

Листинг 2.2. Содержимое файла TypeScript main.ts

```
import {Component} from '@angular/core';
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { platformBrowserDynamic } from '@angular/platform-
➥ browser-dynamic';

// Компонент
@Component({
  selector: 'hello-world',

  template: '<h1>Hello {{ name }}!</h1>'
})

class HelloWorldComponent {
  name: string;

  constructor() {
    this.name = 'Angular';
  }
}
```

Импортирует метод инициализации и аннотацию @Component из соответствующих пакетов Angular, что делает их доступными для кода приложения

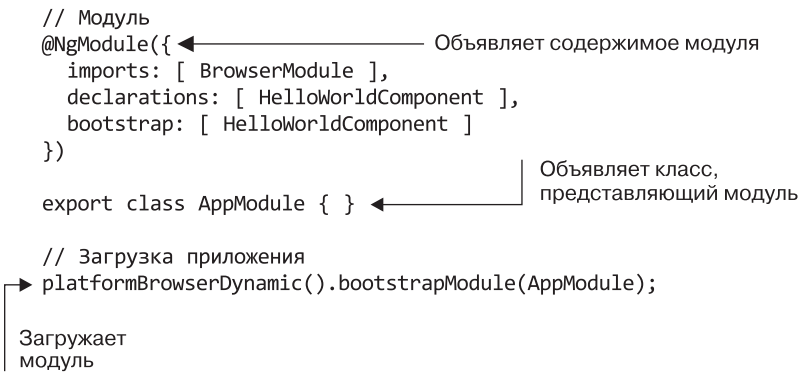
Аннотация @Component, помещенная над классом HelloWorldComponent, превращает этот класс в компонент Angular

Свойство template определяет разметку HTML для отрисовки этого компонента

Аннотированный класс HelloWorldComponent представляет собой компонент

Свойство name используется в выражении привязки данных для шаблона компонента

Внутри конструктора вы инициализируете свойство name с помощью значения, которое Angular2 связал с шаблоном



Мы познакомимся с аннотациями `@Component` и `@NgModule` в разделе 2.2.

Что такое метаданные?

Как правило, метаданные — это пояснительная информация о данных. Например, в MP3-файле музыка является данными, а имя исполнителя, название песни и обложка альбома — метаданные. MP3-плеер включает в себя обработчик метаданных, который читает метаданные и отображает некую их часть во время воспроизведения песни.

Когда речь идет о классах, метаданные — дополнительная информация о классе. Например, декоратор `@Component` (также известный как «аннотация») указывает Angular (обработчику метаданных), что это не обычный класс, а компонент. Angular генерирует дополнительный код JavaScript, основываясь на информации, предоставленной в свойствах декоратора `@Component`.

Если же мы говорим о свойствах классов, декоратор `@Input` указывает Angular, что это свойство класса должно поддерживать привязку и иметь возможность получать данные из родительского компонента.

Кроме того, можно рассматривать декоратор как функцию, прикрепляющую некоторые данные к декорированному элементу. Декоратор `@Component` не изменяет декорированный класс, а лишь добавляет данные, описывающие его, чтобы компилятор Angular мог сгенерировать финальную версию кода компонента либо в памяти браузера (динамическая компиляция), либо в файле на диске (статическая компиляция).

Любой элемент приложения можно включить в файл HTML (или шаблон другого элемента) путем использования тега, который соответствует имени компонента в свойстве `selector` аннотации `@Component`. Селекторы компонентов похожи на селекторы CSS, поэтому, учитывая наличие селектора `'hello-world'`, вы отрисуете данный элемент на странице HTML с помощью элемента с именем `<hello-world>`. Angular преобразует эту строку в `document.querySelectorAll(selector)`.

Обратите внимание на то, как в листинге 2.2 весь шаблон заключен в обратные кавычки для того, чтобы преобразовать его в строку. Таким образом, вы можете указывать одинарные и двойные кавычки внутри шаблона и разбивать его на несколько строк для более удачного форматирования. Шаблон содержит выражение для привязки данных `{{ name }}`, и во время выполнения Angular будет определять свойство `name` вашего компонента и заменять выражение привязки данных, располагающееся в фигурных скобках, конкретным значением.

Вы будете применять TypeScript для всех примеров кода, приведенных в этой книге, за исключением двух версий приложения Hello World, показанных далее. В одном из примеров будет приведена версия, написанная на ES5, а в другом — на ES6.

2.1.2. Hello World с помощью ES5

Для создания приложений на ES5 вы должны использовать особый пакет Angular, поставляющийся в формате Universal Module Definition (UMD, универсальное определение модуля) (обратите внимание на сочетание *umd*, присутствующее в URL). Этот пакет публикует все API Angular в глобальном объекте `ng`. Файл HTML приложения Hello World, написанного с помощью ES5, может выглядеть вот так (см. каталог `hello-world-es5`) (листинг 2.3).

Листинг 2.3. Содержимое файла ES5 index.html

```
<!DOCTYPE html>
<html>
<head>
  <script src="//unpkg.com/zone.js@0.6.12/dist/zone.js"></script>
  <script src="//unpkg.com/rxjs@5.0.0-beta.11/bundles/Rx.umd.js"></script>
  <script src="//unpkg.com/core-js/client/shim.min.js"></script>
  <script src="//unpkg.com/@angular/core@2.0.0/bundles/core.umd.js"></script>
  <script src="//unpkg.com/@angular/common@2.0.0/bundles/common.umd.js">
   ➥ </script>
  <script src="//unpkg.com/@angular/compiler@2.0.0/bundles/
    compiler.umd.js"></script>
  <script src="//unpkg.com/@angular/platform-browser@2.0.0/bundles/
   ➥ platform-browser.umd.js"></script>
  <script src="//unpkg.com/@angular/platform-browser-dynamic@2.0.0/
bundles/platform-browser-dynamic.umd.js"></script>
</head>
<body>
<hello-world></hello-world>
<script src="main.js"></script>
</body>
</html>
```

Поскольку ES5 не поддерживает аннотации и не имеет нативной системы модулей, файл `main.js` должен отличаться от версии для TypeScript (листинг 2.4).

Листинг 2.4. Содержимое файла ES5 main.js

```
// Компонент
(function(app) {
  app.HelloWorldComponent =
      ng.core.Component({
        selector: 'hello-world',
        template: '<h1>Hello {{name}}!</h1>'
      })
      .Class({
        constructor: function() {
          this.name = 'Angular 2';
        }
      });
})(window.app || (window.app = {}));

// Модуль
(function(app) {
  app.AppModule =
      ng.core.NgModule({
        imports: [ ng.platformBrowser.BrowserModule ],
        declarations: [ app.HelloWorldComponent ],
        bootstrap: [ app.HelloWorldComponent ]
      })
          .Class({
            constructor: function() {}
          });
})(window.app || (window.app = {}));

// Инициализация приложения
(function(app) {
  document.addEventListener('DOMContentLoaded', function() {
    ng.platformBrowserDynamic
        .platformBrowserDynamic()
        .bootstrapModule(app.AppModule);
  });
})(window.app || (window.app = {}));
```

Первая функция-выражение, вызываемая сразу после создания (immediately invoked function expression, IIFE), вызывает методы `Component()` и `Class` глобального пространства имен Angular `ng.core`. Вы определяете объект `HelloWorldComponent`, а метод `Component` прикрепляет метаданные, определяющие селектор и шаблон. Это позволяет превратить объект JavaScript в визуальный элемент.

Бизнес-логика компонента располагается в методе `Class`. В данном случае вы объявляете и инициализируете свойство `name`, которое будет привязано к шаблону компонента.

Вторая IIFE вызывает метод `NgModule` для создания модуля, который определяет объект `HelloWorldComponent` и указывает, что он является основным компонентом, присваивая его имя свойству `bootstrap`. Наконец, третья IIFE запускает приложение, вызывая метод `bootstrapModule()`, который загружает модуль, создает объект `HelloWorldComponent` и прикрепляет его к DOM браузера.

2.1.3. Hello World с помощью ES6

Версия ES6 приложения Hello World очень похожа на версию TypeScript, но в качестве компилятора SystemJS в этом случае используется Traceur. Основной файл `index.html` выглядит так (листинг 2.5).

Листинг 2.5. Содержимое файла ES6 index.html

```
<!DOCTYPE html>
<html>
<head>
  <script src="//unpkg.com/zone.js@0.6.21"></script>
  <script src="//unpkg.com/reflect-metadata@0.1.3"></script>
  <script src="//unpkg.com/traceur@0.0.111/bin/traceur.js"></script>
  <script src="//unpkg.com/systemjs@0.19.37/dist/system.src.js"></script>
  <script>
    System.config({
      transpiler: 'traceur',
      traceurOptions: {annotations: true},
      map: {
        'rxjs': 'https://unpkg.com/rxjs@5.0.0-beta.12',

        '@angular/core' : 'https://unpkg.com/@angular/
          ➥ core@2.0.0',
        '@angular/common' : 'https://unpkg.com/@angular/
          ➥ common@2.0.0',
        '@angular/compiler' : 'https://unpkg.com/@angular/
          ➥ compiler@2.0.0',
        '@angular/platform-browser' : 'https://unpkg.com/@angular/
          ➥ platform-browser@2.0.0',
        '@angular/platform-browser-dynamic': 'https://unpkg.com/@angular/
          ➥ platform-browser-dynamic@2.0.0'
      },
      packages: {
        '@angular/core' : {main: 'index.js'},
        '@angular/common' : {main: 'index.js'},
        '@angular/compiler' : {main: 'index.js'},
        '@angular/platform-browser' : {main: 'index.js'},
        '@angular/platform-browser-dynamic': {main: 'index.js'}
      }
    });
    System.import('main.js');
  </script>
</head>
<body>
  <hello-world></hello-world>
</body>
</html>
```

Es6 поддерживается не во всех браузерах, поэтому вам нужно использовать Traceur для того, чтобы скомпилировать (в браузере) код, написанный с помощью ES6, в версию Es5

Файл сценария теперь имеет расширение .js

Единственное различие между файлами `main.js` для ES6 и `main.ts` для TypeScript заключается в том, что во втором случае вам не нужно иметь заранее объявленный член класса `name` (листинг 2.6).

Листинг 2.6. Содержимое файла ES6 main.js

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

// Компонент
@Component({
  selector: 'hello-world',
  template: '<h1>Hello {{ name }}!</h1>'
})
class HelloWorldComponent {
  constructor() {
    this.name = 'Angular 2';
  }
}

// Модуль
@NgModule({
  imports: [ BrowserModule ],
  declarations: [ HelloWorldComponent ],
  bootstrap: [ HelloWorldComponent ]
})
export class AppModule { }

// Инициализация приложения
platformBrowserDynamic().bootstrapModule(AppModule);
```

2.1.4. Запуск приложений

Чтобы запустить любое веб-приложение, вам нужен простой сервер HTTP, например, http-сервер или live-сервер. Последний позволяет автоматически перезагружать веб-страницы по мере модификации кода и сохранения файла запущенного приложения.

Для установки http-сервера используйте следующую команду `npm`:

```
npm install http-server -g
```

Для запуска сервера из командной строки в корневом каталоге проекта примените такую команду:

```
http-server
```

Нам нужно, чтобы сервер выполнял автоматические перезагрузки в браузере, поэтому установим и запустим live-сервер, используя аналогичную процедуру:

```
npm install live-server -g
live-server
```

Если вы задействуете http-сервер, то понадобится вручную открыть браузер и ввести URL http://localhost:8080, а live-сервер откроет браузер за вас.

Чтобы запустить приложение Hello World, запустите live-сервер в корневом каталоге проекта; он загрузит в ваш браузер страницу `index.html`. Вы должны увидеть на странице надпись Hello Angular 2! (рис. 2.1). В браузере на панели Developer Tools (Инструменты разработчика) вы можете увидеть, что шаблон, указанный вами для `HelloWorldComponent`, становится содержимым элемента `<hello-world>`. При этом выражение привязки данных заменяется реальным значением, которое вы использовали для инициализации свойства `name` в конструкторе компонента.

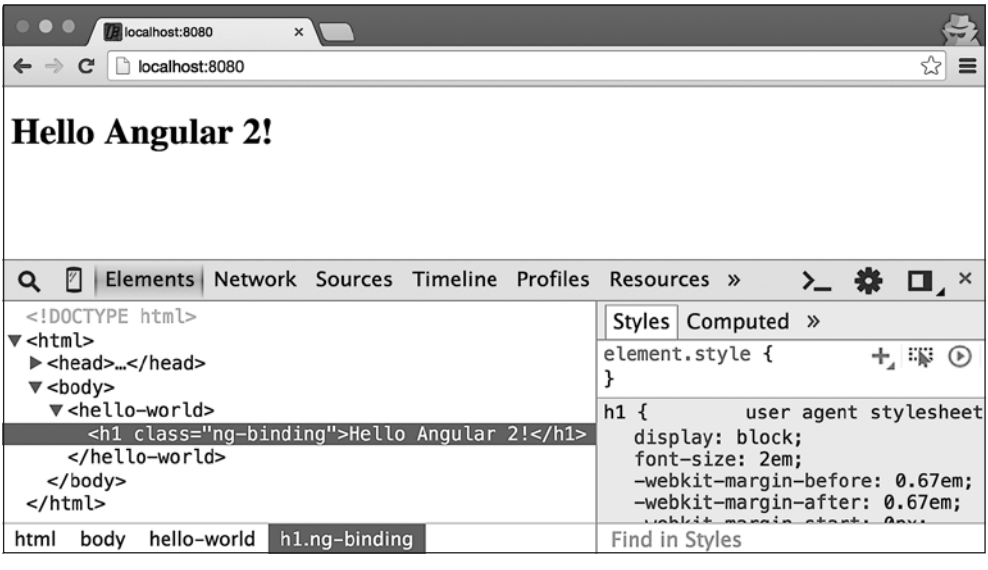


Рис. 2.1. Запуск приложения Hello World

2.2. Элементы Angular-приложения

В данном разделе мы рассмотрим основные составные части приложений, написанных с помощью Angular, чтобы вы смогли читать и понимать Angular-код. Мы разберем каждую из этих тем более подробно в следующих главах.

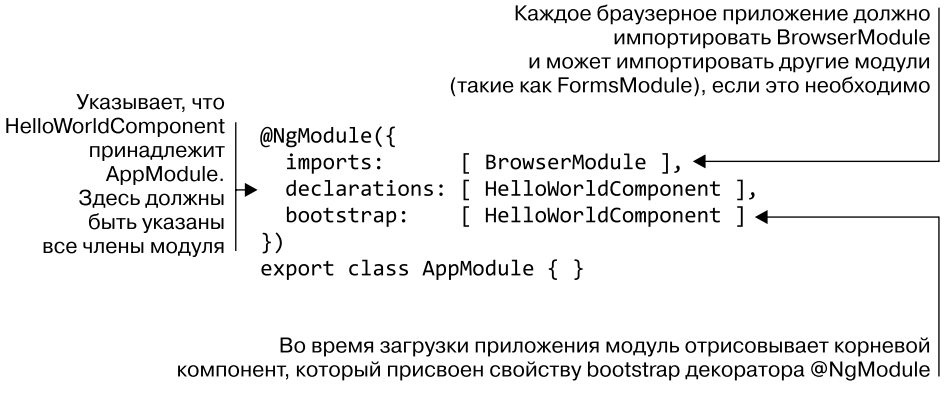
2.2.1. Модули

Модуль Angular представляет собой контейнер для группы связанных элементов, сервисов, директив и т. д. Модуль можно считать библиотекой компонентов и сервисов, реализующей определенную функциональность в основном домене вашего приложения, например, модуль отправки товара или модуль формирования счетов. Все элементы небольшого приложения могут находиться в одном модуле (корневом), а более крупные приложения — иметь больше одного модуля (модуль с функциональностью). Все приложения должны иметь хотя бы один корневой модуль, который инициализируется во время запуска приложения.

ОБРАТИТЕ ВНИМАНИЕ

Модули ES6 предлагают способ скрыть и защитить функции или переменные и создать загружаемые сценарии. Модули Angular, напротив, используются для упаковки связанной с приложением функциональности.

С точки зрения синтаксиса модуль — это класс, аннотированный декоратором `NgModule`, который может содержать другие ресурсы. В разделе 2.1 вы уже использовали модуль, выглядевший вот так:



Импорт `BrowserModule` обязателен в корневом модуле, но, если ваше приложение будет иметь корневой модуль и модули с функциональностью, последние вместо него должны будут импортировать `CommonModule`. Члены всех импортированных модулей (например, `FormsModule` и `RouterModule`) доступны всем компонентам модуля.

Чтобы загрузить и скомпилировать модуль при запуске приложения, вам нужно вызвать модуль `bootstrapModule`:

```
platformBrowserDynamic().bootstrapModule(AppModule);
```

Модули вашего приложения могут быть загружены либо немедленно, как показано в предыдущем фрагменте кода, либо лениво (по мере необходимости) — это делает маршрутизатор (см. главу 3). Декоратор `@NgModule` встречается в каждой главе данной книги, так что вы сможете увидеть, как объявлять модули, имеющие несколько членов. Для получения более подробной информации о модулях Angular прочтите документацию на https://angular.io/guide/ngmodule.

2.2.2. Компоненты

Это основной элемент Angular-приложения. Каждый из них состоит из двух частей: представления, определяющего пользовательский интерфейс, и класса, реализующего логику, лежащую за представлением.

Любое Angular-приложение — это иерархия компонентов, упакованных в модули. Приложение должно иметь хотя бы один модуль и хотя бы один элемент, который

называется *корневым*. Он ничем не отличается от других. Любой компонент, присвоенный свойству `bootstrap` вашего модуля, становится корневым.

Чтобы создать компонент, объявите класс и прикрепите к нему аннотацию `@Component`:

```
@Component({
  selector: 'app-component',
  template: '<h1>Hello !</h1>'
})
class HelloComponent {}
```

Каждая аннотация `@Component` должна задавать свойства `selector` и `template` (или `templateUrl`), которые позволяют установить, как компонент должен определяться и отрисовываться на странице.

Свойство `selector` похоже на селекторы CSS. Каждый элемент HTML, соответствующий селектору, отрисовывается как компонент Angular. Декоратор `@Component` можно рассматривать как функцию конфигурирования, которая дополняет класс. Если вы взглянете на скомпилированный код файла `main.ts` из листинга 2.2, то увидите, что компилятор Angular делает с декоратором `@Component`:

```
var core_1;
var HelloWorldComponent;

HelloWorldComponent = (function () {
      function HelloWorldComponent() {
          this.name = 'Angular 2';
      }
      HelloWorldComponent = __decorate([
          core_1.Component({
              selector: 'hello-world',
              template: '<h1>Hello {{ name }}!</h1>'
          }),
          __metadata('design:paramtypes', [])
      ], HelloWorldComponent);
      return HelloWorldComponent;
})
```

Каждый компонент должен определять представление, которое указывается либо в свойстве `template`, либо в свойстве `templateUrl` декоратора `@Component`:

```
@Component({
  selector: 'app-component',
  template: '<h1>App Component</h1>' })
class AppComponent {}
```

Для веб-приложений шаблон содержит разметку HTML. Можно также использовать другой язык разметки для отрисовки нативных мобильных приложений, предоставляемый сторонними фреймворками. Если разметка содержит несколько десятков линий (или меньше), то можно не выносить ее в отдельный файл с помощью свойства `template`. Мы не применяли обратные галочки в предыдущем примере, поскольку он включает всего одну строку разметки и не содержит одинарных

и двойных кавычек. Более крупные фрагменты разметки HTML должны располагаться в отдельном файле HTML, на который ссылается свойство `templateUrl`.

Стиль компонентов задается путем обычного CSS. Вы можете использовать свойство `styles` для встроенных CSS и `styleUrls`, чтобы сослаться на внешний файл со стилями. Внешние файлы позволяют веб-дизайнерам работать со стилями, не изменяя код самого приложения. В конечном счете выбирать между HTML и CSS придется именно вам.

Представление можно рассматривать как результат объединения макета пользовательского интерфейса с данными. Фрагмент кода для `AppComponent` не имеет данных, которые можно объединить, но версия приложения Hello World, написанная с помощью TypeScript (см. файл `main.ts` в листинге 2.2), объединит разметку HTML со значением переменной `name` для того, чтобы создать представление.

ПРИМЕЧАНИЕ

В Angular отрисовка представления откреплена от компонентов, поэтому шаблон может представлять собой характерный для платформы нативный интерфейс наподобие NativeScript (https://www.nativescript.org) или React Native (https://facebook.github.io/react-native).

2.2.3. Директивы

Декоратор `@Directive` позволяет задать свой вариант поведения для элемента HTML (например, можно добавить возможность автозаполнения для элемента `<input>`). Каждый компонент, по сути, является директивой, с которой связано представление, но, в отличие от компонента, директива не имеет собственного представления.

В следующем примере показана директива, которая может быть прикреплена к элементу `input` для того, чтобы записывать входные значения в консоль браузера по мере его изменения:

```
@Directive({
  selector: 'input[log-directive]',
  host: {
    '(input)': 'onInput($event)'
  }
})
class LogDirective {
  onInput(event) {
    console.log(event.target.value);
  }
}
```

Для этого селектора нужно, чтобы целевой элемент HTML имел элемент input и атрибут log-directive

Элемент host — тот элемент, к которому вы прикрепляете директиву

Обработчик элемента <input> записывает свое значение в консоль

Чтобы привязать события к обработчикам, заключите имя события в скобки. Когда в элементе `host` произойдет событие `input`, будет вызван обработчик `onInput()`, и объект события будет передан в этот метод как аргумент.

Рассмотрим пример того, как можно прикрепить директивы к элементу HTML:

```
<input type="text" log-directive/>
```

В следующем примере показана директива, которая изменяет цвет фона прикрепленного элемента на синий:

```
import { Directive, ElementRef, Renderer } from '@angular/core';

@Directive({ selector: '[highlight]' })
export class HighlightDirective {
  constructor(renderer: Renderer, el: ElementRef) {
    renderer.setElementStyle(el.nativeElement, 'backgroundColor', 'blue');
  }
}
```

Эта директива может быть прикреплена к различным элементам HTML, ее конструктор получает ссылки на отрисовщик и элементы пользовательского интерфейса, внедренные с помощью Angular.

Рассмотрим, как можно прикрепить директиву к элементу HTML `<h1>`:

```
<h1 highlight>Hello World</h1>
```

Все директивы, использованные в модуле, должны быть добавлены к свойству `declaration` декоратора `@NgModule`, как показано в этом примере:

```
@NgModule({
  imports: [ BrowserModule ],
  declarations: [ HelloWorldComponent,
                  HighlightDirective ],
  bootstrap: [ HelloWorldComponent ]
})
```

2.2.4. Краткое введение в привязку данных

Angular имеет механизм, который называется *привязкой данных*. Он позволяет синхронизировать свойства компонента и представление. Этот механизм довольно сложен, подробнее мы рассмотрим его в главе 5. Здесь же разберем наиболее распространенные формы синтаксиса привязки данных.

Чтобы отобразить в шаблоне значение как строку, используйте двойные фигурные скобки:

```
<h1>Hello {{ name }}!</h1>
```

Добавьте квадратные скобки, чтобы привязать свойство элемента HTML к значению:

```
<span [hidden]="isValid">The field is required</span>
```

Привязать обработчик события к событию элемента помогут круглые скобки:

```
<button (click)="placeBid()">Place Bid</button>
```

Если вы хотите сослаться на свойство объекта DOM внутри шаблона, то добавьте локальную переменную шаблона (ее имя должно начинаться с символа #), которая автоматически сохранит ссылку на соответствующий объект DOM, и примените точечную нотацию:

```
<input #title type="text" />
<span>{{ title.value }}</span>
```

Теперь, когда вы знаете, как писать простые приложения с помощью Angular, рассмотрим, как можно загрузить в браузер код, используя библиотеку SystemJS.

2.3. Универсальный загрузчик модулей SystemJS

Большинство существующих веб-приложений загружают файлы JavaScript на страницу HTML с помощью тегов `<script>`. Несмотря на то, что можно добавить на страницу код Angular точно таким же способом, мы рекомендуем использовать библиотеку SystemJS. Angular тоже применяет ее для внутренних нужд.

В этом разделе мы кратко рассмотрим данную библиотеку, чтобы вы могли начать разрабатывать приложения на Angular. Для получения более подробного руководства по SystemJS обратитесь к странице библиотеки на GitHub: https://github.com/systemjs/systemjs.

2.3.1. Обзор загрузчиков модулей

В финальной версии спецификации ES6 вводится понятие модулей, а также рассматриваются их синтаксис и семантика (http://www.ecma-international.org/ecma-262/6.0/#sec-modules). В первых набросках данной спецификации вводилось понятие глобального объекта `System`, ответственного за загрузку модулей в исполняемую среду независимо от того, чем она является — браузером или отдельным процессом. Однако от определения этого объекта избавились в финальной версии спецификации ES6. В данный момент он отслеживается группой Web Hypertext Application Technology Working Group (см. http://whatwg.github.io/loader). Объект System может стать частью спецификации ES8.

Загрузчик модулей — полифилл — из спецификации ES6 (https://github.com/ModuleLoader/es-module-loader) предлагает начать использовать объект `System` прямо сейчас (не дожидаясь появления будущих спецификаций EcmaScript). Он стремится соответствовать стандартам будущего, но эта версия полифилла поддерживает только модули ES6.

Поскольку спецификация ES6 относительно нова, большая часть сторонних пакетов, размещенная в реестре NPM, еще не задействует модули ES6. В первых девяти главах данной книги мы применяем SystemJS, который не только включает в себя ES6 Module Loader, но и позволяет загружать модули, написанные в форматах AMD, CommonJS, UMD и других глобальных форматах. Поддержка этих

форматов полностью прозрачна для пользователей SystemJS, поскольку загрузчик автоматически определяет формат модуля, использующий целевой сценарий. В главе 10 вы обратитесь к другому загрузчику модулей, называемому Webpack.

2.3.2. Загрузчики модулей против тегов <script>

Зачем вообще использовать загрузчики модулей, если можно просто проставить теги `<script>`?

У такого подхода есть несколько недостатков.

- Разработчик отвечает за поддержку тегов `<script>` в файлах HTML. Некоторые из этих тегов могут стать избыточными с течением времени, но если вы забудете убрать их, то браузер все еще будет загружать их, увеличивая время загрузки и тратя впустую пропускную способность сети.
- Зачастую имеет значение порядок, в котором загружаются сценарии. Браузеры могут сохранить порядок загрузки сценариев, если только вы поместите теги `<script>` в разделе HTML-документа `<head>`. Однако такой подход считается признаком дурного тона, поскольку не дает отрисовывать страницу до того, как загрузятся все сценарии.

Рассмотрим преимущества использования загрузчиков модулей во время как разработки, так и подготовки рабочей версии вашего приложения.

- В средах разработки код обычно разбивается на несколько файлов, и каждый файл представляет собой модуль. Когда вы импортируете модуль в свой код, загрузчик соотнесет имя модуля с файлом, загрузит этот файл в браузер, а затем выполнит остальную часть кода. Модули позволяют лучше организовывать проекты; загрузчик модулей автоматически собирает все воедино в браузере, когда вы запускаете приложение. Если модуль имеет зависимости, то они также будут загружены.
- Когда вы подготавливаете рабочую версию приложения, загрузчик модулей берет основной файл, проходит по дереву, состоящему из всех модулей, к которым можно получить доступ из него, и объединяет их в один пакет. Таким образом, пакет содержит только код, действительно используемый приложением. Кроме того, это позволяет решить проблему порядка загрузки сценариев и возникновения циклических ссылок.

Указанные преимущества доступны не только для вашего кода, но и для сторонних пакетов (таких как Angular).

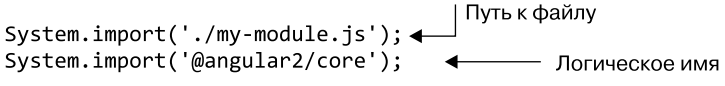
ПРИМЕЧАНИЕ

В книге мы используем термины «модуль» и «файл» попеременно. Модуль не может располагаться в нескольких файлах. Пакет обычно представлен одним файлом и содержит несколько модулей. Если в каком-то конкретном месте имеется разница между модулем и файлом, то мы явно укажем на это.

2.3.3. Приступаем к работе с SystemJS

Когда вы используете SystemJS на странице HTML, эта библиотека становится доступной в качестве глобального объекта `System`, который имеет несколько статических методов. В основном вы будете применять два метода: `System.import()` и `System.config()`.

Чтобы загрузить модуль, задействуйте вызов `System.import()`, принимающий в качестве аргумента имя модуля. Оно может быть как путем к файлу, так и логическим именем, соотнесенным с путем к файлу:



Если имя модуля начинается с символов `./`, то оно является путем к файлу даже при отсутствии у имени расширения. SystemJS поначалу пытается соотнести имя модуля и сконфигурированное соотнесение, предоставленное либо как аргумент метода `System.config()`, либо в файле (наподобие `systemjs.config.js`).

Если имя соотнести нельзя, то оно считается путем к файлу.

ПРИМЕЧАНИЕ

> В этой книге мы будем использовать как префикс ./, так и конфигурацию соотнесения , чтобы определить, какой файл будет загружен. Если вы увидите конструкцию System.import('app') и не сможете найти файл с именем app.ts, то взгляните на конфигурацию соотнесения вашего проекта.

Метод `System.import()` мгновенно возвращает объект `Promise` (см. приложение А). Когда разрешается промис для объекта модуля, после загрузки модуля вызывается метод обратного вызова `then()`. Если промис отклоняется, то ошибки обрабатываются в методе `catch()`.

Объект модуля ES6 содержит свойство для каждого экспортированного значения в загруженном модуле. В следующем фрагменте кода из двух файлов показывается, как можно экспортировать переменную в модуле и использовать ее в другом сценарии:

```
// lib.js
export let foo = 'foo';
// main.js
System.import('./lib.js').then(libModule => {
  libModule.foo === 'foo'; // true
});
```

Здесь вы используете метод `then()`, чтобы указать функцию обратного вызова, которая будет вызвана при загрузке `lib.js`. Загруженный объект передается как аргумент выражения со стрелкой.

В сценариях ES5 вы задействуете метод `System.import()` для загрузки кода либо немедленно, либо лениво (динамически). Например, если ваш сайт посетил анонимный пользователь, то вам, скорее всего, не нужен модуль, реализующий

функциональность пользовательского профиля. Но как только пользователь авторизуется, вы можете динамически загрузить этот модуль, таким образом снижая исходный размер загружаемой страницы и время загрузки.

А как насчет операторов импорта ES6? В вашем первом Angular-приложении вы использовали вызов `System.import()` в файле `index.html` для загрузки корневого модуля приложения, `main.ts`. В свою очередь, сценарий `main.ts` импортирует модули Angular с помощью собственного оператора `import`.

Когда SystemJS загружает `main.ts`, он автоматически компилирует его в код, совместимый с ES5, поэтому в коде, исполняемом браузером, вы не встретите операторов импорта. В будущем, когда модули ES6 будут нативно поддерживаться основными браузерами, данный шаг перестанет быть обязательным, и операторы импорта будут работать аналогично `System.import()` за исключением того, что не смогут контролировать момент, когда загружается модуль.

ПРИМЕЧАНИЕ

Когда загрузчик модулей SystemJS компилирует файлы, он автоматически генерирует отображение исходного кода для каждого файла с расширением .js, что позволяет выполнять отладку кода TypeScript в браузере.

Пример приложения

Рассмотрим приложение, которое должно загружать сценарии ES5 и ES6. Это приложение будет содержать три файла (см. каталог `systemjs-demo`):

```
├── index.html
├── ES6module.js
└── es5module.js
```

В обычном веб-приложении файл `index.html` будет содержать теги `<script>`, ссылающиеся на файлы `ES6module.js` и `es5module.js`. Каждый из этих файлов в таком случае был бы автоматически загружен и запущен в браузере. Но данный подход имеет несколько недостатков, которые мы рассмотрели в подразделе 2.3.2. Изучим способы решения этих проблем с помощью SystemJS в рамках приложения-примера.

Вы используете оператор экспорта ES6, чтобы сделать имя модуля `ES6module.js` доступным за пределами сценария. Наличие оператора экспорта автоматически сделает файл модулем ES6:

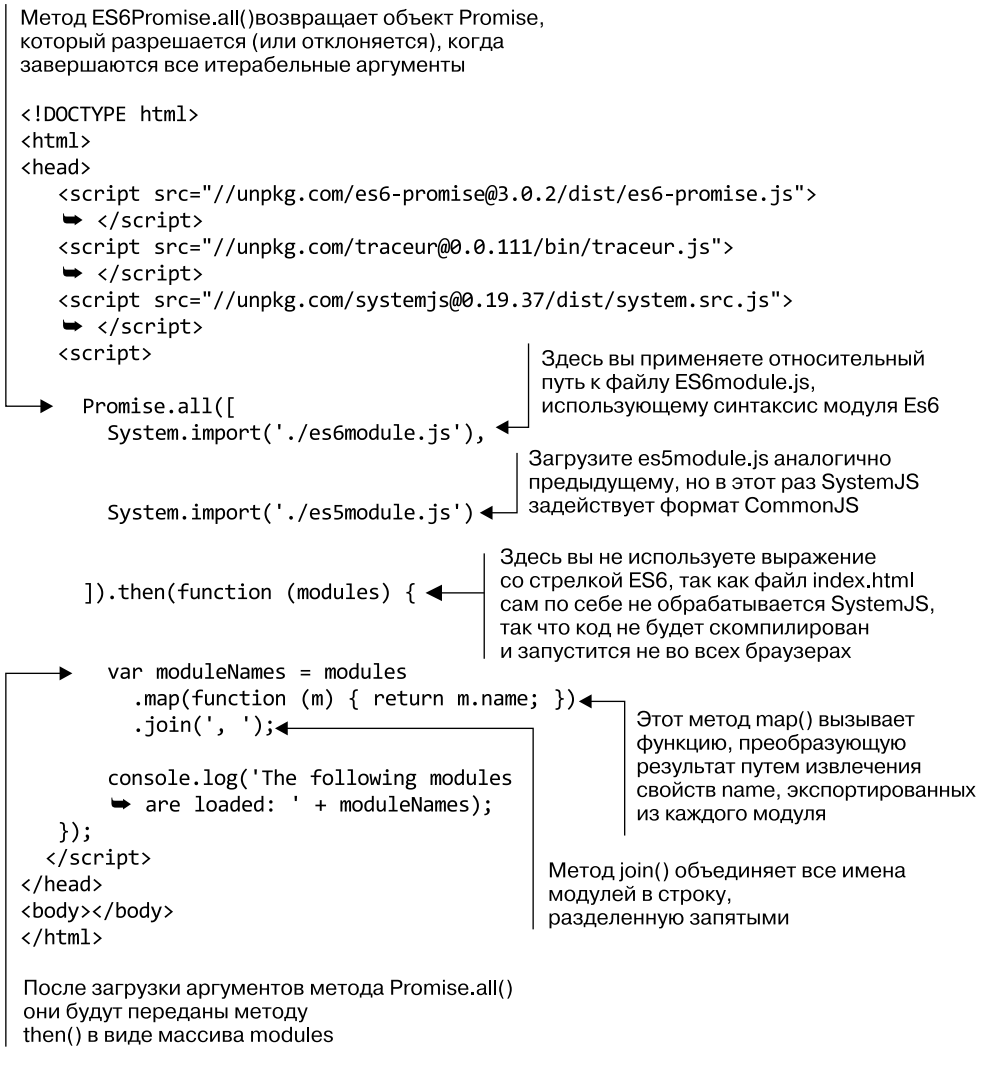
```
export let name = 'ES6';
console.log('ES6 module is loaded');
```

Файл `es5module.js` не содержит синтаксиса ES6 и применяет формат модуля CommonJS для экспортирования имени модуля. По сути, вы прикрепляете к объекту `exports` переменные, которые хотели бы увидеть за пределами модуля:

```
exports.name = 'ES5';
console.log('ES5 module is loaded');
```

В показанном далее файле `index.html` (листинг 2.7) выполняется незаметный импорт модулей CommonJS и ES6 с помощью SystemJS.

Листинг 2.7. Файл index.html и SystemJS



Поскольку вызов `System.import()` возвращает объект `Promise`, вы можете начать загружать по несколько модулей за раз и выполнять другой код, когда все модули загружены.

После запуска приложения вы увидите в консоли разработчика следующий результат (держите открытой панель Developer Tools (Инструменты разработчика), чтобы его увидеть):

```
Live reload enabled.
ES6 module is loaded
ES5 module is loaded
The following modules are loaded: ES6, ES5
```

Первая строка приходит от live-сервера, а не от приложения. Как только будет загружен один из модулей, вы увидите сообщение в журнале. После загрузки всех модулей будет выполнена функция обратного вызова, и вы увидите последнее сообщение журнала.

Конфигурирование систем

До сих пор вы использовали стандартную конфигурацию SystemJS, но можно сконфигурировать практически любой аспект его работы, задействуя метод `System.config()`, принимающий в качестве аргумента объект конфигурации. Данный метод может быть вызван несколько раз с передачей ему разных объектов конфигурации. Если значение одного и того же параметра устанавливается несколько раз, то будет применено последнее значение. Вы можете либо встроить сценарий с помощью `System.config()` в файл HTML, используя тег `<script>` (см. раздел 2.1), либо сохранить код для `System.config()` в отдельном файле (таком как `systemjs.config.js`) и включить его в файл HTML, применяя тег `<script>`.

Полный список параметров конфигурации SystemJS можно найти на GitHub (https://github.com/systemjs/systemjs/blob/master/docs/config-api.md). Мы лишь кратко обсудим некоторые варианты конфигурации, использованные в данной книге.

BASEURL

Все модули загружаются относительно этого URL, если только имя модуля не представляет собой абсолютный или относительный URL:

```
System.config({ baseURL: '/app' });
System.import('es6module.js');   // GET /app/es6module.js
System.import('./es6module.js'); // GET /es6module.js
System.import('http://example.com/es6module.js'); // GET http://example.com/
➥ ES6module.js
```

DEFAULTJSEXTENSIONS

Если `defaultJSExtensions` имеет значение `true`, то расширение `.js` будет автоматически добавлено к пути файла. При наличии у имени модуля расширения, отличного от `.js`, это расширение будет добавлено в любом случае:

```
System.config({ defaultJSExtensions: true });
System.import('./es6module');    // GET /es6module.js
System.import('./es6module.js'); // GET /es6module.js
System.import('./es6module.ts'); // GET /es6module.ts.js
```

ПРИМЕЧАНИЕ

Свойство defaultJSExtensions обеспечивает лишь обратную совместимость, оно будет удалено в будущих версиях SystemJS.

MAP

Этот параметр позволяет создавать псевдоним для имени модуля. Когда вы импортируете модуль, его имя будет заменено связанным значением, если только оригинальное имя не представляет какой-либо путь (абсолютный или относительный). Параметр `map` применяется перед `baseURL`:

```
System.config({ map: { 'es6module.js': 'esSixModule.js' } });
System.import('es6module.js');   // GET /esSixModule.js
System.import('./es6module.js'); // GET /es6Module.js
```

Рассмотрим еще один пример использования параметра `map`:

```
System.config({
  baseURL: '/app',
  map: { 'es6module': 'esSixModule.js' }
});
System.import('es6module'); // GET /app/esSixModule.js
```

PACKAGES

Этот параметр предоставляет удобный способ задать метаданные и соотнести конфигурацию, которая характерна для заданного пути. Например, в следующем фрагменте кода для SystemJS указывается, что вызов `System.import('app')` должен загружать модуль, расположенный в файле `main_router_sample.ts`, путем предоставления лишь имени файла и стандартного расширения для TypeScript — `ts`:

```
System.config({
  packages: {
    app: {
      defaultExtension: "ts",
      main: "main_router_sample"
    }
  }
});
System.import('app');
```

PATHS

Этот параметр похож на `map`, но поддерживает подстановочные символы. Применяется после `map`, но перед `baseURL` (см. листинг 2.6). Вы можете использовать оба этих параметра, но помните, что `paths` является частью спецификации Loader (см. http://whatwg.github.io/loader) и реализации загрузчика модулей ES6 (см. https://github.com/ModuleLoader/es-module-loader), а `map` распознается только SystemJS:

```
System.config({
  baseURL: '/app',
  map: { 'es6module': 'esSixModule.js' },
  paths: { '*': 'lib/*' }
});
System.import('es6module'); // GET /app/lib/esSixModule.js
```

Во многих примерах данной книги вы найдете вызов `System.import('app')`, открывающий файл с другим именем (которое отличается от `app`), поскольку было сконфигурировано свойство `map` или `packages`. Когда вы видите конструкцию наподобие `import {Component} from '@angular/ core';`, `@angular` ссылается на имя, соотнесенное с реальным каталогом, где располагается фреймворк Angular. Элемент `core` является подкаталогом, основной файл в нем указан в конфигурации SystemJS, как продемонстрировано в этом примере:

```
packages: {
        '@angular/core' : {main: 'index.js'}
}
```

TRANSPILER

Данный параметр позволяет указать имя модуля компилятора, который будет использован при загрузке модулей приложения. Если файл не содержит хотя бы один оператор импорта или экспорта, то не будет скомпилирован. Параметр `transpiler` может иметь одно из следующих значений: `typescript`, `traceur` или `babel`:

```
System.config({
  transpiler: 'traceur',
  map: {
    traceur: '//unpkg.com/traceur@0.0.108/bin/traceur.js'
  }
});
```

TYPESCRIPTOPTIONS

Этот параметр позволяет настраивать параметры компилятора TypeScript.

Список всех доступных параметров можно найти в документации к TypeScript: http://www.typescriptlang.org/docs/handbook/compiler-options.html.

2.4. Выбираем менеджер пакетов

Скорее всего, при написании веб-приложения вы будете использовать сторонние библиотеки. В примерах кода, приведенных в нашей книге, тоже применяются несколько сторонних библиотек. Вы будете задействовать фреймворк Angular в большинстве примеров кода данной книги, а для приложения-аукциона вы также обратитесь к библиотеке Bootstrap от Twitter, зависимостью которой является jQuery. Ваше приложение может требовать наличия определенных версий этих зависимостей.

Загрузкой библиотек, фреймворков и их зависимостей управляет менеджер пакетов, и вам нужно решить, на каком из наиболее популярных вариантов остановиться. Разработчики, использующие JavaScript, могут быть ошеломлемы разнообразием доступных менеджеров пакетов: npm, Bower, jspm, Jam и Duo — и это лишь немногие из них.

Типичный проект включает в себя конфигурационный файл, содержащий имена и версии требуемых библиотек и фреймворков. Рассмотрим фрагмент конфигурации `package.json` npm, которую вы будете использовать для приложения-аукциона:

```
"scripts": {
    "start": "live-server"
  },
  "dependencies": {
    "@angular/common": "2.0.0",
    "@angular/compiler": "2.0.0",
    "@angular/core": "2.0.0",
    "@angular/forms": "2.0.0",
    "@angular/http": "2.0.0",
    "@angular/platform-browser": "2.0.0",
    "@angular/platform-browser-dynamic": "2.0.0",
    "@angular/router": "3.0.0",

    "core-js": "^2.4.0",
    "rxjs": "5.0.0-beta.12",
    "systemjs": "0.19.37",
    "zone.js": "0.6.21",
    "bootstrap": "^3.3.6",
    "jquery": "^2.2.2"
    },
    "devDependencies": {
    "live-server": "0.8.2",
    "typescript": "^2.0.0"
  }
```

В разделе `scripts` указывается, какие команды нужно запустить, если вы введете в командной строке `npm start`. В этом случае следует запустить live-сервер. В разделе `dependencies` перечислены все сторонние библиотеки и инструменты, необходимые для работы среды выполнения, в которой будет развернуто приложение.

В разделе `devDependencies` указываются инструменты, которые должны присутствовать на вашем компьютере. Например, вы не будете использовать live-сервер для коммерческой версии, поскольку этот сервер слишком простой и удовлетворяет лишь нужды разработчиков. В приведенной выше конфигурации также указывается: компилятор TypeScript нужен только во время разработки; вы можете догадаться, что весь код TypeScript будет скомпилирован в код JavaScript.

В приведенной конфигурации дается и номер версии. Символ ^ перед ним указывает на то, что для проекта требуется либо указанная версия этой библиотеки или пакета, либо пакет с более новой дополнительной версией. Когда мы использовали бета-версию Angular, мы хотели указывать точную версию пакета, поскольку более новые версии могли значительно отличаться от предыдущих.

В начале работы с Angular мы знали, что будем применять загрузчик модулей SystemJS. Позже нам стало известно, что автор SystemJS (Гай Бедфорд, Guy Bedford) также создал менеджер пакетов jspm, задействующий данный загрузчик, поэтому мы решили использовать jspm. Какое-то время мы применяли npm для установки инструментов, а jspm — для установки зависимостей. Такой подход

работал хорошо, но при использовании jspm браузер выполнял 400+ запросов к серверу для демонстрации первой страницы довольно простого приложения. Нам показалось, что 3,5 секунды, за которые оно открывается, — это довольно много.

Мы решили попробовать применять npm для управления зависимостями во время разработки. Получилось гораздо лучше — всего 30 запросов к серверу, приложение запускалось всего за 1,5 секунды.

Мы кратко рассмотрим оба менеджера пакетов и покажем, как запустить новый проект с помощью каждого из них. Менеджер jspm еще довольно молод и со временем может улучшиться, но для наших проектов мы решили использовать npm.

2.4.1. Сравниваем npm и jspm

Менеджер пакетов npm изначально создавался для управления модулями Node.js, написанными в формате CommonJS. Этот формат не был разработан для веб-приложений, поскольку предполагалось, что модули загружаются синхронно. Рассмотрим следующий фрагмент кода:

```
var x = require('module1');
var y = require('module2');
var z = require('module3');
```

Загрузка модуля `module2` не начнется, пока не будет загружен модуль `module1`, а модуль `module3` запустится только после загрузки `module2`. Это приемлемо для приложений для ПК, написанных с помощью Node.js, поскольку загрузка выполняется с локального компьютера, но подобный синхронный процесс замедлит загрузку приложений.

Еще одним слабым местом npm было использование вложенных зависимостей. Если пакеты A и B зависели от пакета C, то каждый из них хранил копию C в своем каталоге, поскольку A и B могли зависеть от разных версий C. Несмотря на то, что это было приемлемо для приложений, написанных на Node.js, такой подход не годился для приложений, загружающихся в браузер. Даже загрузка одной и той же версии библиотеки дважды может вызвать проблемы. Если же загружены две разные версии, то шанс возникновения сбоя в приложении становится еще выше.

С вложенными зависимостями разобрались в версии npm 3, но проблема была решена лишь частично. По умолчанию, npm пытается установить пакет C в тот же каталог, где находятся A и B, поэтому A и B пользуются одной копией C. Но если A и B требуют наличия конфликтующих версий C, то npm возвращается к подходу с вложенными зависимостями. Библиотеки, созданные для приложений, работающих на стороне клиента, обычно включают встроенные версии (пакеты, состоящие из одного файла) в свои пакеты npm. Пакеты не содержат сторонних зависимостей, в связи с чем нужно вручную загружать их на странице. Это помогает избежать проблемы с вложенными зависимостями.

Менеджер пакетов jspm создан для модулей и загрузчиков модулей ES6. Он не размещает пакеты сам по себе, а предлагает концепцию реестров, которая позволяет создавать собственные каталоги для пакетов. Сразу после установки jspm

дает возможность устанавливать пакеты как из реестра npm, так и непосредственно из репозиториев GitHub.

Менеджер пакетов jspm был разработан для совместной работы с SystemJS. Когда вы инициализируете новый проект или устанавливаете пакет с помощью jspm, он автоматически создает конфигурацию для SystemJS для загрузки модулей. В отличие от npm он использует подход *плоских* зависимостей, так что в проекте имеется только одна копия каждой библиотеки. Это позволяет применять операторы импорта даже для загрузки стороннего кода. Таким образом решается проблема порядка загрузки сценариев и гарантируется, что приложение загружает только те модули, которые будет задействовать на самом деле.

Пакеты jspm обычно не включают другие пакеты. Вместо этого они сохраняют оригинальную структуру проекта и файлы, в связи с чем каждый модуль можно загрузить отдельно. Несмотря на то, что наличие оригинальной версии файла может повысить качество отладки, оно не всегда полезно. Отдельный импорт каждого модуля приводит к загрузке в браузер сотен файлов еще до запуска приложения. Данное обстоятельство замедляет разработку и не подходит для развертывания на производстве.

Еще одно слабое место jspm — вы не можете использовать пакеты npm или репозиторий GitHub как пакет jspm сразу после установки. Для этого может потребоваться дополнительное конфигурирование, чтобы jspm мог соответствующим образом настроить SystemJS так, чтобы модули загружались из пакета.

На момент написания данной книги в реестре jspm имеется меньше 500 пакетов, готовых к работе с SystemJS, по сравнению с 250 000 пакетами для npm.

2.4.2. Создаем проект Angular с помощью npm

Чтобы начать новый проект, управляемый npm, создайте новый каталог (например, `angular-seed`) и откройте его в командном окне. Затем запустите команду `npm init -y`, которая создаст исходную версию конфигурационного файла `package.json`. Как правило, при запуске команды `npm init` вам будет задано несколько вопросов при создании файла, но флаг `-y` позволяет принять значения по умолчанию для всех параметров. В следующем примере показана эта команда, запущенная в пустом каталоге `angular-seed`.

```
$ npm init -y
Wrote to /Users/username/angular-seed/package.json:
{
  "name": "angular-seed",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Большая часть сгенерированной конфигурации нужна либо для публикации проекта в реестре npm, либо для установки пакета в качестве зависимости другого проекта. Вы будете использовать npm только для управления зависимостями проекта и автоматизации процессов разработки и сборки.

Поскольку вы не будете публиковать пакет в реестре npm, следует удалить все свойства, кроме `name`, `description` и `scripts`. Добавьте также свойство `"private": true`, поскольку оно не создается по умолчанию. Это гарантирует, что пакет не будет опубликован в реестре npm. Файл `package.json` должен выглядеть так (листинг 2.8).

Листинг 2.8. Содержимое файла package.json

```
{
  "name": "angular-seed",
  "description": "An initial npm-managed project for Chapter 2",
  "private": true,
    "scripts": {
  "test": "echo \"Error: no test specified\" && exit 1"
  }
}
```

Конфигурация `scripts` позволяет указать команды, доступные для запуска в командном окне. По умолчанию `npm init` создает тестовую команду, которую можно запустить с помощью следующего вызова: `npm test`. Заменим ее командой `start`; ее вы будете использовать для запуска live-сервер, установленного в подразделе 2.4.1. Рассмотрим конфигурацию свойства `scripts`:

```
{
  ...
  "scripts": {
    "start": "live-server"
  }
}
```

Можно запустить любую команду npm из раздела `scripts` с помощью синтаксиса `npm run mycommand`, например, так: `npm run start`. Кроме того, можно использовать сокращение `npm start` вместо `npm run start`. Такой синтаксис доступен только для заранее определенных сценариев npm (см. документацию к npm на https://docs.npmjs.com/misc/scripts).

Теперь нужно, чтобы npm загрузил в этот проект Angular в качестве зависимости. В версии приложения Hello World, написанной с помощью TypeScript, вы применяли код Angular, который размещен на сервере unpkg CDN, но в нашем случае нужно загрузить его в каталог проекта. Вам понадобятся также локальные версии SystemJS, live-сервер и компилятор TypeScript.

Пакеты npm зачастую состоят из других пакетов, оптимизированных для коммерческого применения; они не включают исходный код библиотек. Добавьте в файл `package.json` раздел, который использует исходный код (но не оптимизированные пакеты) заданных пакетов. Этот раздел вставляется сразу после

строки с лицензиями (следует обновить версии зависимостей до самых свежих) (листинг 2.9).

Листинг 2.9. Использование исходного кода пакетов

```
"dependencies": {
  "@angular/common": "2.0.0",
  "@angular/compiler": "2.0.0",
  "@angular/core": "2.0.0",
  "@angular/forms": "2.0.0",
  "@angular/http": "2.0.0",
  "@angular/platform-browser": "2.0.0",
  "@angular/platform-browser-dynamic": "2.0.0",
  "@angular/router": "3.0.0",

  "core-js": "^2.4.0",
  "rxjs": "5.0.0-beta.12",
  "systemjs": "0.19.37",
  "zone.js": "0.6.21"
},
"devDependencies": {
  "live-server": "0.8.2",
  "typescript": "^2.0.0"
}
```

Теперь запустите команду `npm install` в командной строке из каталога, где находится ваш файл `package.json`, и npm начнет загружать предыдущие пакеты и их зависимости в каталог `node_modules`. По завершении этого процесса вы увидите десятки подкаталогов в каталоге `node_modules`, включая `@angular`, `systemjs`, `live-server` и `typescript`:

```
angular-seed
├── index.html
├── package.json
└── app
│   └── app.ts
├── node_modules
│   ├── @angular
│   ├── systemjs
│   ├── typescript
│   ├── live-server
│   └── ...
```

Создадим несколько измененную версию файла `index.html` со следующим содержимым в каталоге `angular-seed` (листинг 2.10).

Листинг 2.10. Обновленный файл index.html

```
<!DOCTYPE html>
<html>
<head>
  <title>Angular seed project</title>
```

```
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">

  <script src="node_modules/typescript/lib/typescript.js"></script>
  <script src="node_modules/core-js/client/shim.min.js"></script>
  <script src="node_modules/zone.js/dist/zone.js"></script>
  <script src="node_modules/systemjs/dist/system.src.js"></script>
  <script src="systemjs.config.js"></script>
  <script>
    System.import('app').catch(function(err){ console.error(err); });
  </script>
</head>

<body>
<app>Loading...</app>
</body>
</html>
```

Обратите внимание на то, что теги `script` теперь загружают требуемые зависимости из локального каталога `node_modules`. То же верно и для конфигурационного файла SystemJS `systemjs.config.js`, показанного далее (листинг 2.11).

Листинг 2.11. Содержимое файла systemjs.config.js

```
System.config({
    transpiler: 'typescript',
    typescriptOptions: {emitDecoratorMetadata: true},
    map: {
      '@angular': 'node_modules/@angular',
      'rxjs' : 'node_modules/rxjs'
    },
    paths: {
      'node_modules/@angular/*': 'node_modules/@angular/*/bundles'
    },
    meta: {
      '@angular/*': {'format': 'cjs'}
    },
    packages: {
      'app'                              : {main: 'main', defaultExtension:
      ➥ 'ts'},
      'rxjs'                             : {main: 'Rx'},
      '@angular/core'                    : {main: 'core.umd.min.js'},
      '@angular/common'                  : {main: 'common.umd.min.js'},
      '@angular/compiler'                : {main: 'compiler.umd.min.js'},
      '@angular/platform-browser'        : {main: 'platform-
      ➥ browser.umd.min.js'},
      '@angular/platform-browser-dynamic': {main: 'platform-browser-
      ➥ dynamic.umd.min.js'}
    }
});
```

Показанная конфигурация SystemJS несколько отличается от представленной в листинге 2.1. Сейчас вы не использовали исходный код пакетов Angular. Вместо этого применили их упакованные и сжатые версии, что позволяет снизить количество сетевых запросов, необходимых для загрузки фреймворка Angular, и данная версия фреймворка занимает меньше места.

Каждый пакет Angular поставляется с каталогом `bundles`, который содержит сжатый код. В разделе `packages` конфигурационного файла `SystemJS` вы соотносите имя `app` с основным сценарием, расположенным в файле `main.ts`, после чего при вызове команды `System.import(app)` в файле `index.html` будет загружаться файл `main.ts`.

Добавьте еще один конфигурационный файл в корневой каталог проекта, где вы указываете параметры компилятора tsc (листинг 2.12).

Листинг 2.12. Содержимое файла tsconfig.json

```
{
  "compilerOptions": {
    "target": "ES5",
    "module": "commonjs",
    "experimentalDecorators": true,
    "noImplicitAny": true
  }
}
```

Если вы не знакомы с TypeScript, то обратитесь к приложению Б, в котором объясняется, что для запуска кода TypeScript его сначала нужно скомпилировать в JavaScript с помощью компилятора tsc.

Фрагменты кода, приведенные в главах 1–7, работают и без явного запуска этого компилятора, поскольку SystemJS сам по себе использует tsc для динамической компиляции TypeScript в JavaScript по мере загрузки файла сценария. Однако вам следует хранить файл `tsconfig.json` в корневом каталоге проекта, поскольку он нужен для некоторых IDE.

ПРИМЕЧАНИЕ

Если код Angular динамически компилируется в браузере (не путать с компиляцией в код на JavaScript), то это называется динамической компиляцией (just-in-time (JIT) compilation). Процесс, когда код скомпилирован заранее и имеет особый заголовок ngc, называется компиляцией перед выполнением (ahead-of-time (AoT) compilation). В данной главе мы опишем приложение, скомпилированное методом JIT.

Код приложения будет находиться в трех файлах:

- `app.component.ts` — единственный компонент вашего приложения;
- `app.module.ts` — объявление модуля, который будет содержать ваш компонент;
- `main.ts` — инициализатор модуля.

В подразделе 2.3.3 вы соотнесли имя с файлом `main.ts`, поэтому создадим каталог `app`, включающий файл `app.component.ts`, который имеет следующее содержимое (листинг 2.13).

Листинг 2.13. Содержимое файла app.component.ts

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

@Component({
    selector: 'app',
    template: `<h1>Hello {{ name }}!</h1>`
})
export class AppComponent {
    name: string;
    constructor() {
        this.name = 'Angular 2';
    }
}
```

Теперь нужно создать модуль, который будет содержать компонент `AppComponent`. Поместите этот код в файл`app.module.ts` (листинг 2.14).

Листинг 2.14. Содержимое файла app.module.ts

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent }  from './app.component';

@NgModule({
    imports:      [ BrowserModule ],
    declarations: [ AppComponent ],
    bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

Этот файл включает лишь определение модуля Angular. Класс имеет аннотацию `@NgModule`, содержащую модуль `BrowserModule`, который должны импортировать все браузеры.

Поскольку в модуле только один класс, нужно указать его в свойстве `declarations` и отметить его как класс `bootstrap`:

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app.module';
platformBrowserDynamic().bootstrapModule(AppModule);
```

Запустите приложение, выполнив команду `npm start` из каталога `angular-seed`. Откроется браузер, и вы на долю секунды увидите сообщение Loading..., за которым последует сообщение Hello Angular 2!. На рис. 2.2 показано, как данное приложение выглядит в браузере Chrome: во вкладке Network (Сеть) открыта панель Developer

Tools (Инструменты разработчика) — это позволяет видеть фрагмент кода, загруженный браузером, а также понадобившееся время.

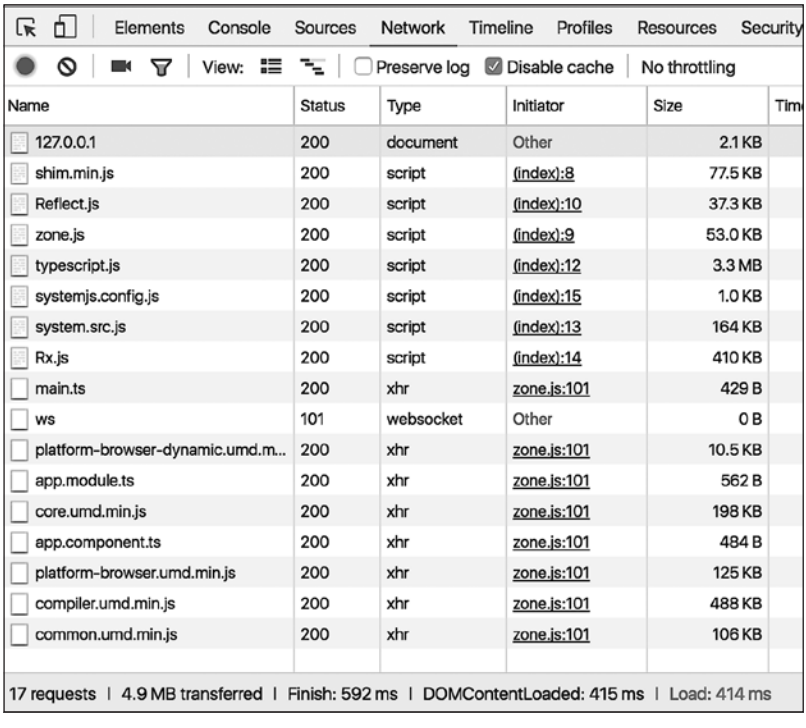


Рис. 2.2. Запуск приложения из проекта, управляемого npm

Не пугайтесь размера загруженного файла, вы оптимизируете его в главе 10. Вы используете live-сервер, поэтому, как только измените и сохраните код приложения, он перезагрузит страницу в соответствии с последней версией кода. Теперь применим полученные вами знания для создания приложения, более сложного, чем Hello World.

2.5. Практикум: приступаем к работе над онлайн-аукционом

С этого момента каждая глава будет заканчиваться разделом «Практикум», в котором содержатся инструкции по разработке определенного аспекта приложения-аукциона. С помощью данного приложения пользователи могут увидеть список рекламируемых товаров, просмотреть более подробную информацию о требуемом продукте, найти продукт и отслеживать ставки других пользователей. Вы будете

постепенно добавлять код в это приложение для того, чтобы закрепить знания, полученные в каждой главе. Исходный код, который поставляется с книгой, включает в себя полную версию раздела «Практикум» для каждой главы (она находится в каталоге `auction`), но мы просим вас попробовать написать код самостоятельно.

В данном упражнении вы настроите среду разработки и создадите исходный макет проекта аукциона. Вы создадите главную страницу, разобьете ее на компоненты Angular и создадите сервис, который позволяет получить продукты. Если вы будете следовать всем инструкциям, приведенным в этом разделе, то страница аукциона будет выглядеть так, как показано на рис. 2.3.

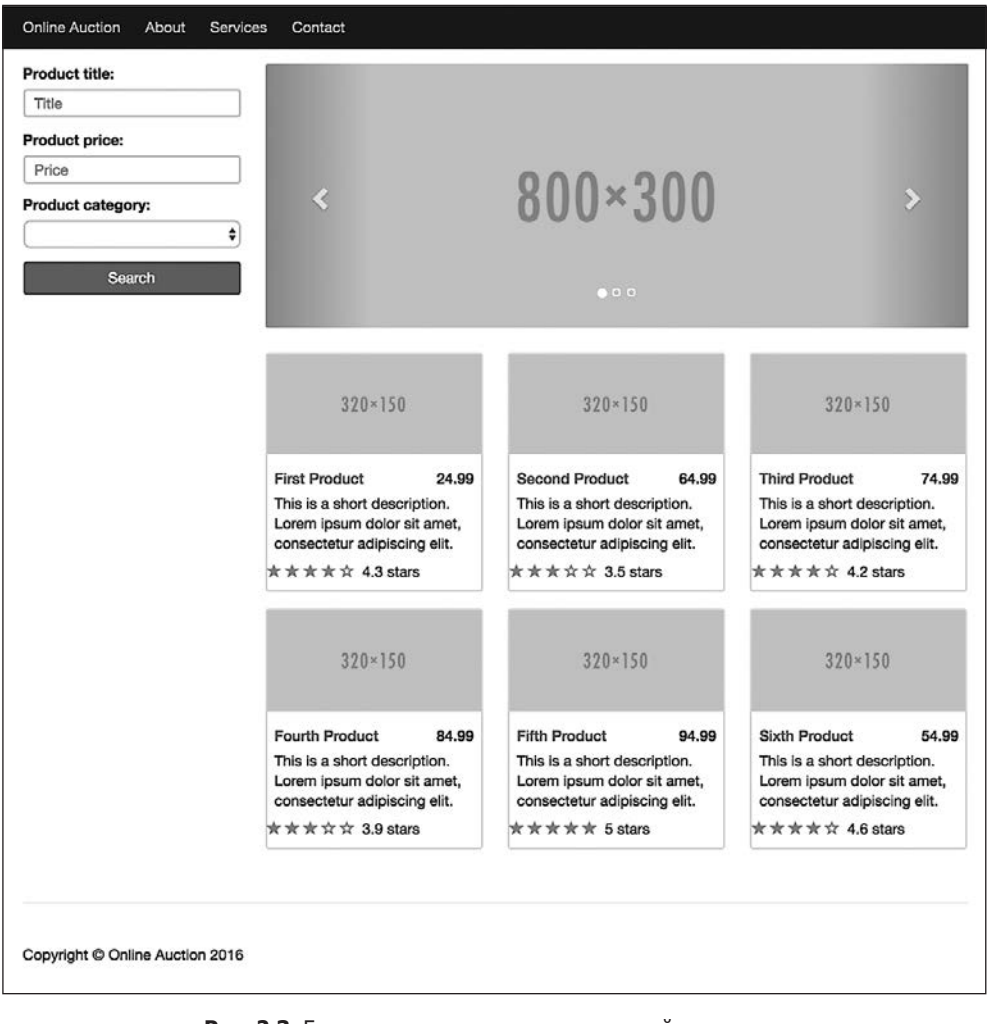


Рис. 2.3. Главная страница приложения онлайн-аукциона

Вы будете использовать серые прямоугольники, предоставленные удобным сервисом Placeholder.com (https://placeholder.com/), генерирующим заполнители нужного размера. Чтобы увидеть сгенерированные изображения, вам нужно подключиться к Интернету во время работы приложения. В следующих разделах содержатся инструкции, которые помогут выполнить упражнение.

ПРИМЕЧАНИЕ

Если вы хотите только прочитать код рабочей версии онлайн-аукциона, то обратитесь к коду, находящемуся в каталоге auction, который поставляется вместе с этой главой. Для запуска предоставленного кода переключитесь в каталог auction, запустите команду npm install, чтобы установить все необходимые зависимости в каталог node_modules, и запустите приложение с помощью команды npm start.

2.5.1. Первичная настройка проекта

Для настройки проекта сначала скопируйте содержимое каталога `angular-seed` в отдельное место и назовите его `auction`. Далее измените поля `name` и `description` в файле `package.json`. Откройте командное окно и переключитесь на только что созданный каталог `auction`. Запустите команду `npm install`; это создаст каталог `node_modules` с зависимостями, указанными в файле `package_json`.

В данном проекте вы будете использовать Twitter Bootstrap в качестве фреймворка CSS, а также библиотеку для создания пользовательского интерфейса, содержащую адаптивные компоненты. Адаптивный веб-дизайн — подход, позволяющий создавать сайты, которые изменяют макет на основе ширины области просмотра устройства пользователя. Термин *адаптивный компонент* означает, что макет элемента может адаптироваться в зависимости от размера экрана. Поскольку библиотека Bootstrap создана на основе jQuery, вам нужно запустить следующие команды для ее установки:

```
npm install bootstrap --save
npm install jquery --save
```

Установите Bootstrap. Параметр --save добавит эту зависимость в файл package.json

Пакет Bootstrap не указывает в качестве своей зависимости jQuery, поэтому нужно устанавливать его отдельно. Такие зависимости называются одноранговыми

СОВЕТ

Мы рекомендуем использовать такие IDE, как WebStorm или Visual Studio Code. Большую часть шагов, требуемых для завершения этого упражнения, можно выполнить с помощью IDE. WebStorm даже позволяет открывать окно терминала внутри IDE.

Теперь создайте файл `systemjs.config.js`, чтобы сохранить конфигурацию SystemJS (листинг 2.15). Вы включите его в тег `<script>` в файле `index.html`.

Листинг 2.15. Содержимое файла systemjs.config.js


```
System.config({
    transpiler: 'typescript',
    typescriptOptions: {emitDecoratorMetadata: true,
     target: "ES5",
     module: "commonjs"},
    map: {
        '@angular': 'node_modules/@angular',
        'rxjs' : 'node_modules/rxjs'
    },
    paths: {
        'node_modules/@angular/*': 'node_modules/@angular/*/bundles'
    },
    meta: {
        '@angular/*': {'format': 'cjs'}
    },
    packages: {
        'app' : {main: 'main',
         ➥ defaultExtension: 'ts'},
        'rxjs' : {main: 'Rx'},
        '@angular/core' : {main: 'core.umd.min.js'},
        '@angular/common' : {main: 'common.umd.min.js'},
        '@angular/compiler' : {main: 'compiler.umd.min.js'},
        '@angular/platform-browser' : {main: 'platform-
         browser.umd.min.js'},
        '@angular/platform-browser-dynamic': {main: 'platform-browser-
         dynamic.umd.min.js'}
    }
});
```

Файл `systemjs.config.js` должен быть включен в `index.html`. Эта конфигурация пакета `app` позволяет использовать строку `<script>System.import ('app')</script>` в файле `index.html`, что загрузит содержимое файла `app/main.ts`.

На этом мы заканчиваем конфигурировать среду разработки для проекта аукциона. Теперь вы готовы писать код приложения.

ПРИМЕЧАНИЕ

На момент написания данной книги команда разработчиков Angular создает компоненты Angular Material (см. https://material.angular.io). Вы можете использовать их вместо Twitter Bootstrap, когда они будут готовы.

2.5.2. Разработка главной страницы

В данном упражнении вы создадите главную страницу; она будет разбита на несколько компонентов Angular. Это упростит поддержку кода и позволит повторно использовать элементы других представлений. На рис. 2.4 вы увидите главную страницу, на которой выделены все компоненты. Вам нужно создать каталоги для хранения всех элементов и сервисов приложения, показанных далее:

```
app
└── components
    ├── application
    ├── carousel
    ├── footer
    ├── navbar
    ├── product-item
    ├── search
    └── stars
└── services
```

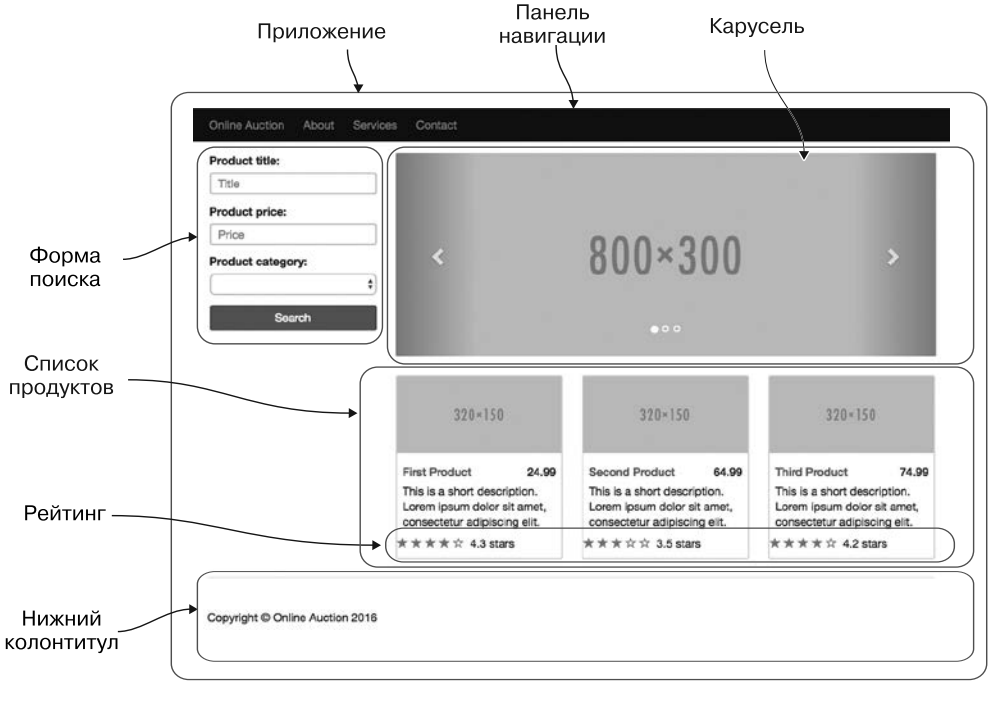


Рис. 2.4. Главная страница онлайн-аукциона, ее компоненты выделены

Каждый каталог внутри каталога `components` содержит код соответствующего компонента. Это позволяет хранить вместе все файлы, связанные с одним элементом. Большинство компонентов состоит из двух файлов: HTML и TypeScript.

Но иногда может понадобиться добавить файл CSS, в котором будут содержаться стили компонента. Каталог `services` будет включать файл с классами, поставляющий данные приложению.

Первая версия главной страницы состоит из семи компонентов. В данном упражнении мы рассмотрим (а вы создадите) три самых интересных элемента, расположенных в каталогах `application`, `product-item` и `stars`. Это позволит попрактиковаться в написании кода, а по завершении упражнения вы можете скопировать остальные компоненты в каталог с вашим проектом.

ПРИМЕЧАНИЕ

В разделе «Практикум» главы 3 вы проведете рефакторинг кода, чтобы интегрировать карусель и продукты в компонент HomeComponent.

Точкой входа в приложение является файл `index.html`. Вы скопировали его из каталога `angular-seed`, и теперь нужно его модифицировать (листинг 2.16). Этот файл довольно мал и увеличится в размерах лишь незначительно, поскольку большая часть ваших зависимостей будет загружена с помощью SystemJS, а весь пользовательский интерфейс представлен одним корневым компонентом Angular, который будет самостоятельно использовать элементы-потомки.

Листинг 2.16. Обновленный файл index.html

```
<!DOCTYPE html>
<html>
<head>
  <title>CH2: Online Auction</title>
  <link rel="stylesheet" href="node_modules/bootstrap/dist/css/
  ➥ bootstrap.css">

  <script src="node_modules/jquery/dist/jquery.min.js"></script>
  <script src="node_modules/bootstrap/dist/js/bootstrap.min.js"></script>

  <script src="node_modules/core-js/client/shim.min.js"></script>
  <script src="node_modules/zone.js/dist/zone.js"></script>

  <script src="node_modules/typescript/lib/typescript.js"></script>
  <script src="node_modules/systemjs/dist/system.src.js"></script>
  <script src="systemjs.config.js"></script>

  <script>
    System.import('app').catch(function (err) {console.error(err);});
  </script>
</head>
<body>
<auction-application></auction-application>
</body>
</html>
```

Добавляет CSS для Bootstrap

Добавляет Bootstrap и jQuery для поддержки компонента-карусели

Загружает main.ts в соответствии с конфигурацией, приведенной в файле systemjs.config.js

Содержимое файла `main.ts` из каталога `app` останется таким же, как и в проекте `angular-seed`:

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app.module';
platformBrowserDynamic().bootstrapModule(AppModule);
```

Обновите файл `app.module.ts`, объявив все компоненты и сервисы, которые будете использовать в приложении (листинг 2.17).

Листинг 2.17. Обновленный файл app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import ApplicationComponent from '. /components/application/application';
import CarouselComponent from "./components/carousel/carousel";
import FooterComponent from "./components/footer/footer";
import NavbarComponent from "./components/navbar/navbar";
import ProductItemComponent from "./components/product-item/product-item";
import SearchComponent from "./components/search/search";
import StarsComponent from "./components/stars/stars";
import {ProductService} from "./services/product-service";

@NgModule({
    imports:      [ BrowserModule ],
    declarations: [ ApplicationComponent,
                    CarouselComponent,
                    FooterComponent,
                    NavbarComponent,
                    ProductItemComponent,
                    SearchComponent,
                    StarsComponent],
    providers:    [ProductService],
    bootstrap:    [ ApplicationComponent ]
})
export class AppModule { }
```

Объявляет все компоненты, которые будет использовать ваш модуль

Объявляет поставщика сервиса ProductService, которого вы чуть позже внедрите в компонент ApplicationComponent

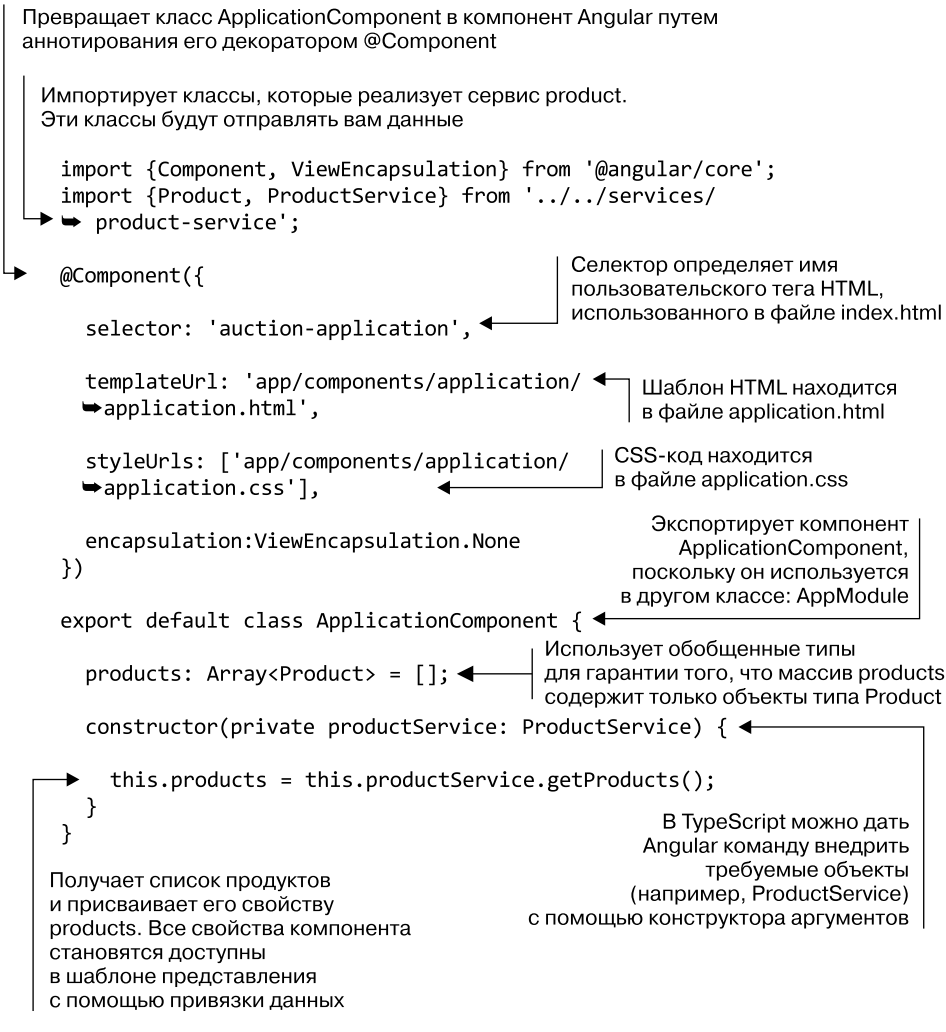
В этом модуле вы объявили все элементы, а также поставщик одного сервиса, который собираетесь создать. Последнее необходимо для механизма внедрения зависимостей. Мы рассмотрим поставщики и внедрение зависимостей в главе 4.

Компонент application

Является корневым компонентом аукциона, таковым он объявлен в `AppModule`. Он служит узлом для всех остальных элементов. Его исходный код состоит из трех файлов: `application.ts`, `application.html` и `application.css`. Мы предполагаем, что вы знакомы с основами CSS, поэтому не будем рассматривать здесь соответствующий файл, а уделим внимание лишь первым двум.

Создайте компонент `ApplicationComponent` и сохраните его в файле `application.ts`, который располагается в каталоге `app/components/application`. Его содержимое показано ниже (листинг 2.18).

Листинг 2.18. Содержимое файла application.ts



Простое объявление аргументов конструктора и их типа даст команду Angular создать и внедрить этот объект (`ProductService`). Внедряемые объекты нужно сконфигурировать с помощью поставщиков, вы уже объявили один поставщик в модуле `AppModule` ранее. Префикс `private` превратит `productService` в переменную — член класса, и вы сможете получить к ней доступ, используя конструкцию `this.productService`.

ПРИМЕЧАНИЕ

В листинге 2.18 используется стратегия инкапсуляции представлений ViewEncapsulation.None, чтобы применять стили из файла application.css не только для компонента ApplicationComponent, но и для всего приложения. Мы рассмотрим различные стратегии инкапсуляции в главе 6.

Создайте файл `application.html` со следующим содержимым (листинг 2.19).

Листинг 2.19. Содержимое файла application.html

```
<auction-navbar></auction-navbar>
<div class="container">
  <div class="row">
    <div class="col-md-3">
      <auction-search></auction-search>
    </div>
    <div class="col-md-9">
      <div class="row carousel-holder">
        <div class="col-md-12">
          <auction-carousel></auction-carousel>
        </div>
      </div>
      <div class="row">
        <div *ngFor="let prod of products" class="col-sm-4 col-lg-4 col-md-4">
          <auction-product-item [product]="prod"></auction-product-item>
        </div>
      </div>
    </div>
  </div>
</div>
<auction-footer></auction-footer>
```

Вы будете использовать несколько собственных элементов HTML, которые представляют ваши компоненты: `<auction-navbar>`, `<auction-search>`, `<auction-carousel>`, `<auction-product-item>` и `<auction-footer>`. Вы добавите их в файл `index.html` точно так же, как и `<auction-application>`.

Самая интересная часть этого файла — способ отображения списка продуктов. Каждый из них будет представлен одинаковым фрагментом HTML на вебстранице. Поскольку у вас есть несколько продуктов, нужно отрисовать один и тот же фрагмент HTML несколько раз. Директива `NgFor` используется внутри компонента для прохождения в цикле по списку элементов коллекции данных и отрисовки разметки HTML для каждого элемента. Вы можете применять укороченную версию синтаксиса `*ngFor`, чтобы представить директиву `NgFor`.

```
<div *ngFor="let prod of products" class="col-sm-4 col-lg-4 col-md-4">
  <auction-product-item [product]="prod"></auction-product-item>
</div>
```

Поскольку `*ngFor` находится внутри тега `<div>`, каждая итерация цикла отрисует `<div>` с содержимым соответствующего элемента `<auction-product-item>`. Чтобы передать объект продукта в компонент `ProductComponent`, используйте квадратные скобки для реализации привязки свойств: `[product]="prod"`. Элемент `[product]` ссылается на соответствующий продукт, расположенный внутри компонента, представленного `<auction-product-item>`, а `prod` — это локальная переменная шаблона, динамически объявленная в директиве `*ngFor` с помощью конструкции `let prod`. Мы рассмотрим привязку свойств более подробно в главе 5.

Стили `col-sm-4`, `col-lg-4` и `col-md-4` мы получаем из библиотеки Bootstrap от Twitter, где ширина окна поделена на 12 невидимых колонок. В нашем примере вы должны выделить 4 колонки (треть ширины элемента `<div>`), если устройство имеет маленький (`sm` (small) — 768 пикселов или больше), большой (`lg` (large) — 1200 пикселов и больше) или средний (`md` (medium) — 992 пиксела или больше) размер экрана.

Поскольку мы не указали количество колонок для очень маленьких устройств (`xs` —768 пикселов или меньше), вся ширина элемента `<div>` будет выделена одному элементу `<auction-product>`.

Для просмотра изменений макета страницы для экранов разных размеров сузьте окно своего браузера так, чтобы его ширина стала меньше, чем 768 пикселов. Вы можете прочесть более подробно о системе сетей библиотеки Bootstrap в документации на http://getbootstrap.com/css/#grid.

ПРИМЕЧАНИЕ

Компонент ApplicationComponent полагается на наличие других элементов (например, ProductItemComponent), которые вы создадите на следующих шагах. Если попробуете запустить аукцион сейчас, то увидите ошибки в консоли разработчика вашего браузера.

Компонент product item

В каталоге `product-item` создайте файл `product-item.ts`, в котором объявляется компонент `ProductItemComponent`, представляющий один объект продукта аукциона (листинг 2.20). Исходный код файла `product-item.ts` структурирован примерно так же, как и код файла `application.ts`: вначале идут операторы импорта, а затем объявление класса компонента, аннотированное декоратором `@Component`.

Листинг 2.20. Содержимое файла product-item.ts

```
import {Component, Input} from '@angular/core';
import StarsComponent from 'app/components/stars/stars';
import {Product} from 'app/services/product-service';

@Component({
  selector: 'auction-product-item',
  templateUrl: 'app/components/product-item/product-item.html'
})
export default class ProductItemComponent {
  @Input() product: Product;
}
```

Свойство компонента `product` аннотировано `@Input()`. То есть значение данного свойства будет доступно родительскому элементу, который может связать с ним значение. Мы рассмотрим свойства для ввода информации более подробно в главе 6.

Создайте файл `product-item.html`, содержащий следующий шаблон компонента `product` (который будет представлен ценой, заголовком и описанием) (листинг 2.21).

Листинг 2.21. Содержимое файла product-item.html

```
<div class="thumbnail">
  <img src="http://placehold.it/320x150">
  <div class="caption">
    <h4 class="pull-right">{{ product.price }}</h4>
    <h4><a>{{ product.title }}</a></h4>
    <p>{{ product.description }}</p>
  </div>
  <div>
    <auction-stars [rating]="product.rating"></auction-stars>
  </div>
</div>
```

Здесь вы используете еще один вид привязки данных: выражение внутри двойных фигурных скобок. Angular оценивает значение выражения внутри скобок, превращает результат в строку и заменяет выражение в шаблоне результирующей строкой. Этот процесс реализован путем интерполяции строк.

Обратите внимание на тег `<auction-stars>`, представляющий компонент `StarsComponent` и объявленный в модуле `AppModule`. Вы привяжете значение свойства `product.rating` к свойству `rating` компонента `StarsComponent`. Чтобы это сработало, рейтинг должен быть объявлен как входное свойство в элементе `StarsComponent`, который вы создадите далее.

Компонент stars

Отображает рейтинг продукта. На рис. 2.5 вы можете увидеть, что он показывает среднее значение рейтинга 4.3, а также значки звездочек, представляющие рейтинг.

Рис. 2.5. Компонент stars

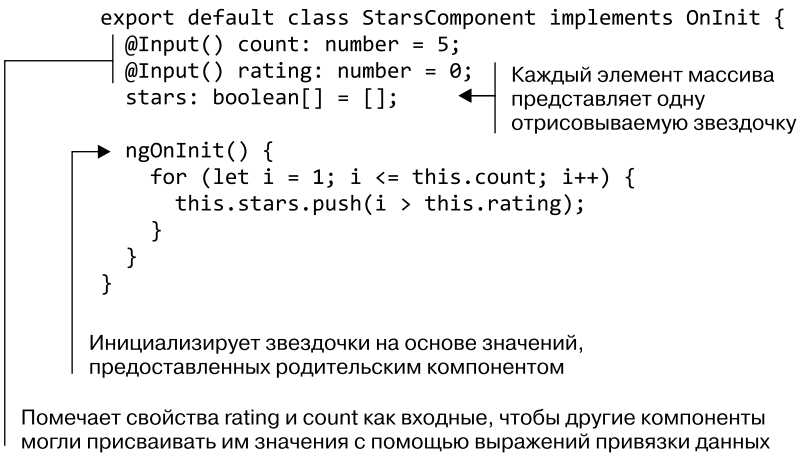
Angular дает возможность воспользоваться привязками жизненного цикла (см. главу 6), позволяющими определить методы обратного вызова, которые будут вызваны в определенные моменты жизненного цикла компонента. В этом элементе вы примените метод обратного вызова `ngOnInit()`, вызываемый при создании компонента и инициализации его свойств.

Создайте файл `stars.ts` со следующим содержимым в каталоге `stars` (листинг 2.22).

Листинг 2.22. Содержимое файла stars.ts

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

@Component({
  templateUrl: 'app/components/stars/stars.html',
  styles: [` .starrating { color: #d17581; }`],
  selector: 'auction-stars'
})
```

Импортирует интерфейс OnInit, где объявляется метод ngOnInit()



Свойство `count` содержит общее количество звездочек, которое требуется отрисовать. Если это свойство не было инициализировано предком, то компонент по умолчанию отрисует пять звезд. Свойство `rating` хранит средний рейтинг, определяющий, сколько звездочек должны быть закрашены, а сколько — остаться пустыми. В массиве `stars` элементы со значением `false` представляют пустые звездочки, а элементы со значением `true` заполнены цветом.

Вы инициализируете массив `stars` в методе обратного вызова `ngOnInit()`, который будет использован в шаблоне для отрисовки звездочек. Метод `ngOnInit()` вызывается всего однажды, сразу после того, как привязанные к данным свойства будут проверены в первый раз, и до того, как будут проверены его потомки. Когда вызывается метод `ngOnInit()`, все свойства, передающиеся от представления-предка, уже инициализированы, поэтому можно применять значение рейтинга для вычисления значений в массиве `stars`.

Кроме того, можно сделать свойство `stars` геттером, чтобы динамически вычислять его значение, но в таком случае это свойство будет вызываться всякий раз, когда Angular синхронизирует модель с представлением. Один и тот же массив будет вычисляться несколько раз.

Создайте шаблон компонента `StarsComponent` в файле `stars.html`, как показано далее (листинг 2.23).

Листинг 2.23. Содержимое файла stars.html

```
<p>
  <span *ngFor="let star of stars"
        class="starrating glyphicon glyphicon-star"
        [class.glyphicon-star-empty]="star">
  </span>
  <span>{{ rating }} stars</span>
</p>
```

Вы уже пользовались директивой `NgFor` и выражением для привязки данных в фигурных скобках в компоненте `ApplicationComponent`. Здесь вы привязываете имя класса CSS к выражению: `[class.glyphicon-star-empty]="star"`. Если правая часть выражения внутри двойных кавычек имеет значение `true`, то класс CSS `glyphicon-star-empty` будет добавлен к атрибуту `class` элемента `<span>`.

Копируем остальной код

Чтобы завершить этот проект, скопируйте недостающие компоненты из каталога `chapter2/auction` в соответствующие каталоги вашего проекта.

- Каталог `services` содержит файл `product-service.ts`, в котором объявляются два класса: `Product` и `ProductService`. Здесь будут храниться данные аукциона. Более подробно содержимое данного файла мы рассмотрим в разделе «Практикум» главы 3.
- Каталог `navbar` содержит код верхней панели навигации.
- Каталог `footer` содержит код нижнего колонтитула страницы.
- Каталог `search` содержит исходный код компонента `SearchComponent`, представляющего собой форму, которую вы разработаете в главе 7.
- Каталог `carousel` содержит код, реализующий ползунок Bootstrap, расположенный в верхней части страницы.

2.5.3. Запуск онлайн-аукциона

Чтобы запустить онлайн-аукцион, откройте командное окно и активируйте live-сервер в каталоге вашего проекта. Вы можете сделать это, введя команду `npm start`, которая сконфигурирована в файле `package.json`. Откроется окно браузера, и вы сможете увидеть главную страницу, как показано на рис. 2.4. Страница, содержащая подробную информацию о продукте, еще не реализована, поэтому ссылки, расположенные в заголовке продукта, не будут работать.

Мы рекомендуем использовать для разработки браузер Chrome, поскольку он предоставляет лучшие инструменты для отладки кода. Откройте панель Developer Tools (Инструменты разработчика) и не закрывайте ее, пока запускаете все фрагменты кода. Если увидите неожиданные результаты, то во вкладке Console (Консоль) сможете найти сообщения об ошибке.

Кроме того, для браузера Chrome существует расширение под именем Augury, позволяющее выполнять отладку приложений Angular. После его установки вы увидите на панели Developer Tools (Инструменты разработчика) дополнительную вкладку Augury (рис. 2.6), которая дает возможность просматривать и изменять значения компонентов вашего приложения во время его работы.

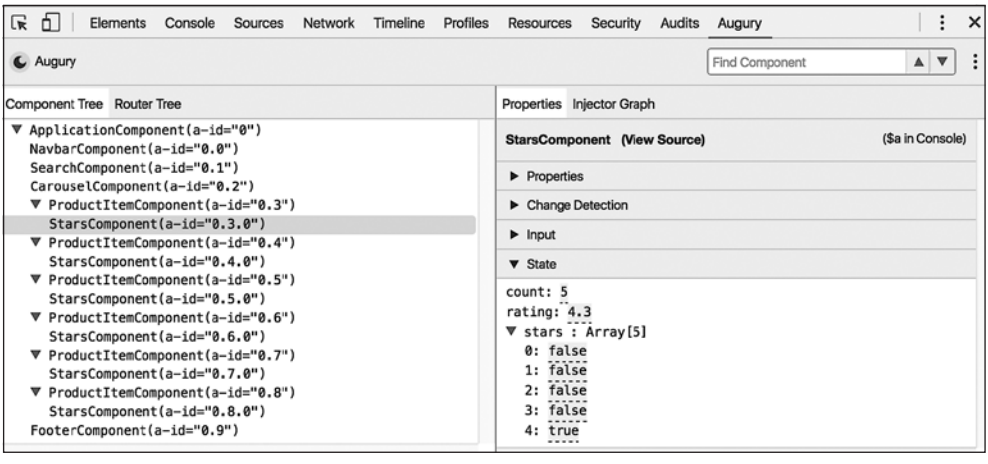


Рис. 2.6. Вкладка Augury

2.6. Резюме

В этой главе вы приступили к разработке Angular-приложения. Мы кратко рассмотрели базовые принципы и основные составные части приложения. В последующих главах мы разберем их более детально. Кроме того, вы создали первую версию приложения онлайн-аукциона, увидели, как настраивается среда разработки, и ознакомились со структурой проекта Angular.

- Angular-приложение представлено иерархией компонентов, которые упаковываются в модули.
- Каждый элемент Angular содержит шаблон для отрисовки пользовательского интерфейса и аннотированный класс, реализующий функциональность этого компонента.
- Шаблоны и стили могут находиться как в одном файле, так и в отдельных.
- Загрузчик модулей SystemJS позволяет разбить приложение на модули ES6 и динамически собрать их воедино во время выполнения.
- Параметры конфигурации для SystemJS могут быть указаны в отдельном конфигурационном файле.
- Использование npm для управления зависимостями — самый простой способ конфигурирования нового проекта Angular.

3 Навигация с помощью маршрутизатора Angular

В этой главе:

- конфигурация маршрутов;
- передача данных при переходе от одного маршрута к другому;
- создание более одной области для навигации (так называемая область отображения) на одной странице с использованием auxiliary;
- ленивая загрузка модулей с помощью маршрутизатора.

В главе 2 вы создали главную страницу онлайн-аукциона, намереваясь создать одностраничное приложение (single-page application, SPA): главная страница не перезагружается целиком, но ее части могут меняться. Теперь нужно добавить навигацию для этого приложения, чтобы область с содержимым изменялась в зависимости от действий пользователя. Ему нужно увидеть подробную информацию о продукте, ставки на продукты, а также пообщаться с продавцами. Маршрутизатор Angular позволяет сконфигурировать и реализовать подобную навигацию без перезагрузки страницы.

Вам нужно не только изменять представление внутри одной страницы, но и поместить в закладки ее URL, чтобы быстрее получать детальную информацию об определенном продукте. Для этого следует присвоить каждому представлению уникальный URL.

Как правило, маршрутизатор можно представить в виде объекта, ответственного за состояние представления приложения. Каждое приложение имеет один маршрутизатор, и вам нужно его сконфигурировать, чтобы он работал.

Сначала мы рассмотрим основные особенности маршрутизатора, а затем вы добавите в приложение-аукцион второе представление — Product Details (Информация о продукте). Когда пользователь выберет какой-то продукт на главной странице, это представление будет отображать подробную информацию о нем.

3.1. Основы маршрутизации

Одностраничные приложения можно рассматривать как коллекцию состояний, например, состояние `Home`, `Product Details` и `Shipping`. Каждое из них является каким-то представлением одного приложения. До этого момента мы работали только с одним представлением состояния: главной страницей.

Онлайн-аукцион (см. рис. 2.4) имеет в верхней части страницы панели навигации (компонент), слева — форму поиска (еще один компонент), в нижней части — нижний колонтитул (еще один компонент), и вам нужно, чтобы все они были видны постоянно. Остальная часть страницы состоит из области содержимого, в которой отображаются элементы `<auction-carousel>` и `<auction-product>`. Вы будете повторно использовать эту область содержимого (*область отображения*) для демонстрации разных представлений в зависимости от действий пользователя.

Для этого нужно сконфигурировать маршрутизатор так, чтобы он мог отображать разные представления в области отображения, *заменяя одно представление другим*. Данная область содержимого представлена тегом `<router-outlet>`. На рис. 3.1 показана область, которую вы будете использовать для отображения разных представлений.

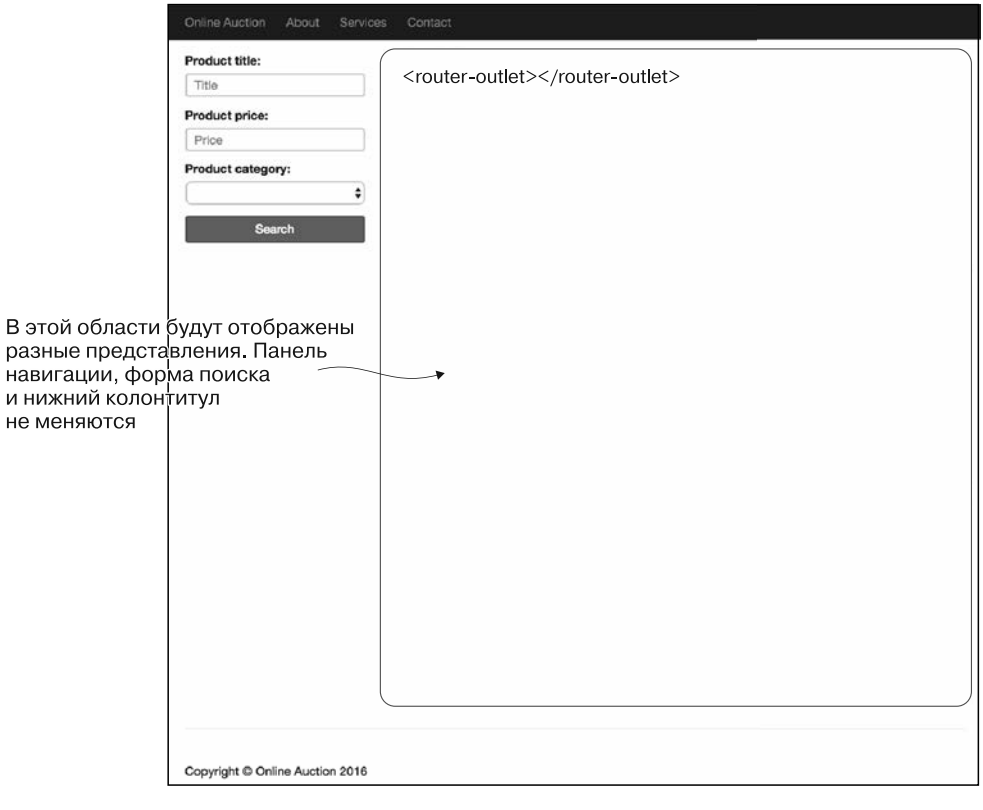


Рис. 3.1. Выделение области для изменения представлений

ПРИМЕЧАНИЕ

На странице может находиться больше одной области отображения. Мы рассмотрим этот вопрос в разделе 3.5.

Вы будете присваивать компоненты для каждого представления, которое хотите отобразить в данной области. В главе 2 вы не создавали родительский компонент, который бы инкапсулировал карусель и продукты аукциона, но к концу этой главы перепишете код, создав `HomeComponent`, — он послужит предком для карусели и продуктов. Кроме того, вы создадите `ProductDetailComponent`; его задача — представлять подробную информацию о каждом продукте. В любой момент времени пользователь будет видеть либо `HomeComponent`, либо `ProductDetailComponent` в области `<router-outlet>`.

Маршрутизатор отвечает за управление навигацией на клиентской стороне, и в подразделе 3.1.2 мы кратко рассмотрим, из чего состоит маршрутизатор. В мире, где отсутствуют одностраничные приложения, навигация по сайту реализуется как набор запросов на сервер, который обновляет всю страницу, отправляя в браузер подходящие документы HTML. В одностраничных приложениях код для отрисовки компонентов уже находится на клиентской стороне (за исключением ситуаций, когда совершается ленивая загрузка), и вам лишь нужно заменить одно представление другим.

По мере того, как пользователь перемещается по приложению, оно может выполнять запросы на сервер для получения или отправки данных. Иногда представление (комбинация кода интерфейса и данных) уже загрузило в браузер все необходимое, но в некоторых случаях оно станет общаться с сервером, отправляя запросы Ajax или используя WebSockets. Каждое представление будет иметь уникальный URL, который демонстрируется в адресной строке браузера. Эту тему мы рассмотрим чуть позже.

В данной области будут отображены разные представления. Панель навигации, форма поиска и нижний колонтитул не меняются.

3.1.1. Стратегии расположения

В любой момент времени адресная строка браузера отображает URL текущего представления. URL может состоять из разных частей (сегментов). Он начинается с протокола, за ним идет доменное имя и иногда номер порта. Параметры, которые нужно передать на сервер, обычно следуют за знаком вопроса (верно для HTTP-запросов `GET`), и выражение может выглядеть так:

```
http://mysite.com:8080/auction?someParam=123
```

Изменение любого символа в предшествующем URL приведет к генерации нового запроса на сервер.

```
&lt;router-outlet&gt;&lt;/router-outlet&gt;
```

В одностраничных приложениях нужно иметь возможность модифицировать URL, не выполняя запрос на сервер, чтобы приложение могло найти подходящее

представление на клиентской стороне. Angular предлагает два варианта реализации навигации на стороне клиента:

- `HashLocationStrategy` — символ решетки (#) добавляется к URL, и сегмент, стоящий после этого символа, уникальным образом определяет маршрут, который будет использован для фрагмента веб-страницы. Такая стратегия работает для всех браузеров, включая старые;
- `PathLocationStrategy` — стратегия, основанная на History API, работает только в браузерах, поддерживающих HTML5. Применяется в Angular по умолчанию.

Навигация с помощью символа решетки

Пример URL, использующий подобную стратегию навигации, показан на рис. 3.2. Изменение любого символа, стоящего справа от решетки, не вызовет отправки запроса на сервер, но позволит перейти к представлению, на которое указывает путь (с параметрами или без) после решетки. Символ решетки служит разделителем между основным URL и локациями с требуемым содержимым на стороне клиента.

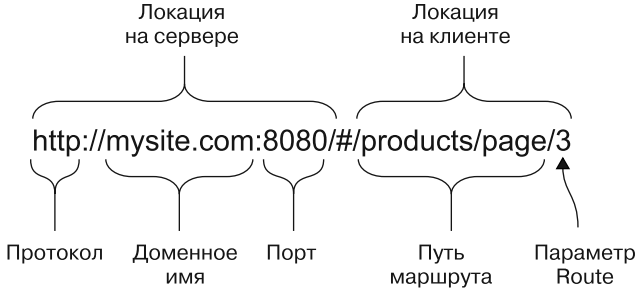


Рис. 3.2. Составные части URL

Попробуйте поперемещаться по одностраничному приложению, такому как Gmail, и проследите за URL. Когда вы находитесь в папке Inbox (Входящие), он выглядит так: https://mail.google.com/mail/u/0/#inbox. Теперь перейдите в папку Sent (Отправленные), фрагмент URL с решеткой изменится с inbox на sent. Код JavaScript клиентской стороны вызывает необходимые функции для отображения представления Sent (Отправленные).

Но почему приложение Gmail демонстрирует сообщение Loading, когда вы переключаетесь на представление Sent (Отправленные)? Код JavaScript этого представления все еще может выполнять запросы Ajax на сервер для того, чтобы получить новые данные, но не загружает дополнительный код, разметку или код CSS.

В данной книге мы будем использовать именно такой способ навигации, а `@NgModule` будет включать следующее значение `providers` (эта тема рассматривается в главе 4):

```
providers:[{provide: LocationStrategy, useClass: HashLocationStrategy}]
```

Навигация, основанная на History API

History API браузера позволяет перемещаться по истории посещений, а также программно манипулировать стеком с историей (см. раздел «Манипуляции с историей браузера» в Mozilla Developer Network, https://developer.mozilla.org/en-US/docs/Web/API/History_API). В частности, метод `pushState()` используется для того, чтобы прикрепить сегмент к основному URL, когда пользователь перемещается по одностраничному приложению.

Рассмотрим следующий URL: http://mysite.com:8080/products/page/3. Сегмент `products/page/3` может быть прикреплен к основному URL программно без использования символа решетки. Если пользователь перейдет со страницы 3 на страницу 4, то код приложения прикрепит к основному URL конструкцию `products/page/4`, сохранив предыдущее состояние `products/page/3` в историю браузера.

Angular избавляет от необходимости вызывать метод `pushState()` явно — нужно лишь сконфигурировать сегменты URL и соотнести их с необходимыми компонентами. Если вы используете стратегию навигации, основанную на History API, то должны указать Angular, что именно является основным URL в вашем приложении, чтобы он мог корректно добавлять сегменты URL клиентской стороны. Вы можете сделать это одним из следующих способов.

- Добавить тег `<base>` в заголовок файла `index.html`, например, `<base href="/">`.
- Присвоить значение константы `APP_BASE_HREF` в корневом модуле и использовать его как значение свойства `providers`. В следующем фрагменте кода в качестве основного URL применяется `/`, но можно задействовать любой сегмент URL, который указывает на конец основного URL:

```
import { APP_BASE_HREF } from '@angular/common';
...
@NgModule({
...
providers:[{provide: APP_BASE_HREF, useValue: '/'}]
})
class AppModule { }
```

3.1.2. Составные части механизма навигации на стороне клиента

Ознакомимся с основными концепциями реализации навигации на стороне клиента с помощью маршрутизатора Angular. Во фреймворке Angular реализация функциональности маршрутизации располагается в отдельном модуле `RouterModule`. Если вашему приложению нужна маршрутизация, то убедитесь, что ваш файл `package.json` содержит зависимость `@angular/router`. Наш файл `package.json` включает следующую строку:

```
"@angular/router": "3.0.0".
```

Помните: цель этой главы — объяснить вам, как организовать навигацию между разными представлениями одностраничного приложения, поэтому для начала нужно сконфигурировать маршрутизатор и добавить его в объявление модуля.

Angular предлагает использовать следующие основные элементы при реализации маршрутизации в приложении.

- `Router` — объект, который представляет собой маршрутизатор во время выполнения программы. Вы можете применять его методы `navigate()` и `navigateByUrl()` для того, чтобы переместиться либо по сконфигурированному пути маршрута, либо по сегменту URL соответственно.
- `RouterOutlet` — директива, служащая заполнителем внутри вашей веб-страницы (`<router-outlet>`), где `router` должен отрисовать элемент.
- `Routes` — массив маршрутов, соотносящих адреса URL и компоненты, которые должны быть отрисованы внутри `<router-outlet>`.
- `RouterLink` — директива для объявления ссылки на маршрут, если навигация выполняется с помощью якорных тегов HTML. Может содержать параметры, которые передаются компоненту маршрута.
- `ActivatedRoute` — объект, представляющий маршрут или маршруты, активные в данный момент.

Маршруты конфигурируются в отдельном массиве объектов типа `Route`. Рассмотрим пример:

```
const routes: Routes = [
    {path: '', component: HomeComponent},
    {path: 'product', component: ProductDetailComponent}
];
```

Поскольку конфигурация маршрута выполняется на уровне модулей, нужно импортировать маршруты в декораторе `@NgModule`. Если вы объявляете маршруты для корневых модулей, то должны использовать метод `forRoot()`, например, так:

```
import { BrowserModule } from '@angular/platform-browser';
import { RouterModule } from '@angular/router';
...
@NgModule({
  imports: [ BrowserModule, RouterModule.forRoot(routes)],
    ...
})
```

Если вы конфигурируете маршруты для модуля функциональности (не для корневого), то используйте метод `forChild()`:

```
import { CommonModule } from '@angular/common';
import { RouterModule } from '@angular/router';
...
```

```
@NgModule({
  imports: [ CommonModule, RouterModule.forChild(routes)],
    ...
})
```

Обратите внимание: в модулях функциональности вы импортируете модуль `CommonModul`, а не `BrowserModule`.

В обычном случае вы будете реализовывать навигацию, выполняя следующие шаги.

1. Сконфигурируйте маршруты для ваших приложений так, чтобы соотнести сегменты URL и соответствующие им компоненты, и передайте объект конфигурации либо методу `RouterModule.forRoot()`, либо `RouterModule.forChild()` в качестве аргумента. Если некоторые компоненты должны принимать входные значения, то можете использовать параметры маршрута.
2. Импортируйте возвращенное значение метода `forRoot()` или метода `forChild()` в декоратор `@NgModule`.
3. Определите область отображения, где маршрутизатор будет отрисовывать компоненты с помощью тега `<router-outlet>`.
4. Добавьте якорные теги HTML, к которым привязаны свойства `[routerLink]` (квадратные скобки указывают на привязку свойств). Теперь, когда пользователь нажмет ссылку, маршрутизатор отрисует соответствующий компонент. Свойство `[routerLink]` можно рассматривать как замену атрибуту `href` якорного тега HTML на клиентской стороне.

Вызов метода маршрутизатора `navigate()` — альтернатива использованию `[routerLink]` для навигации по маршруту. В любом из этих случаев маршрутизатор найдет соответствующий предоставленному пути компонент, создаст (или найдет) объект указанного компонента и исходя из этого обновит URL.

Рассмотрим пример приложения, иллюстрирующий эти шаги (его можно найти в каталоге `router_samples`). Предположим, вы хотите создать в верхней части страницы корневой компонент, который имеет две ссылки, `Home` и `Product Details`. Приложение должно отрисовывать один из этих компонентов в зависимости от того, какую ссылку нажмет пользователь.

Компонент `HomeComponent` будет отрисовывать текст `Home Component` на красном фоне, а `ProductDetailComponent` — текст `Product Details Component` на голубом. Изначально веб-страница должна отображать компонент `HomeComponent`, как показано на рис. 3.3. После того, как пользователь выберет ссылку `Product Details`, маршрутизатор должен отобразить компонент `ProductDetailComponent`, как показано на рис. 3.4.

Основная цель данного упражнения заключается в том, чтобы познакомить вас с маршрутизатором, поэтому элементы будут очень простыми. Рассмотрим код компонента `HomeComponent` (листинг 3.1).

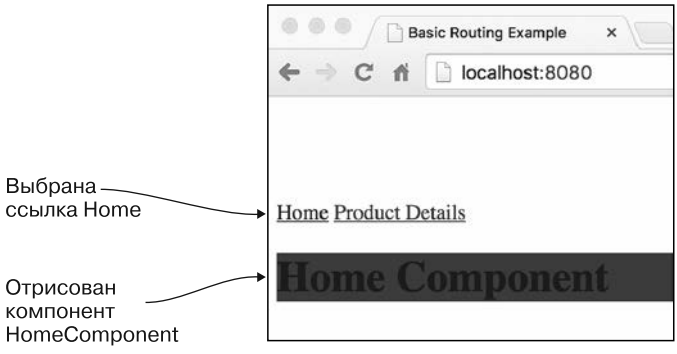


Рис. 3.3. Маршрут Home приложения basic_routing

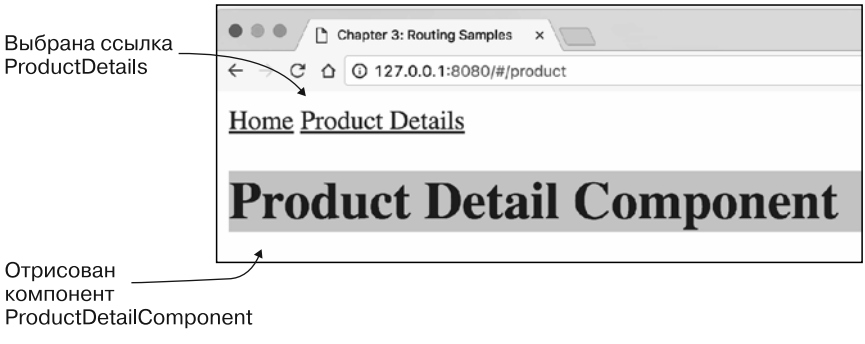


Рис. 3.4. Маршрут Product Details приложения basic_routing

Листинг 3.1. Код компонента HomeComponent

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

@Component({
    selector: 'home',
    template: '<h1 class="home">Home Component</h1>',
    styles: ['.home {background: red}']})
export class HomeComponent {}
```

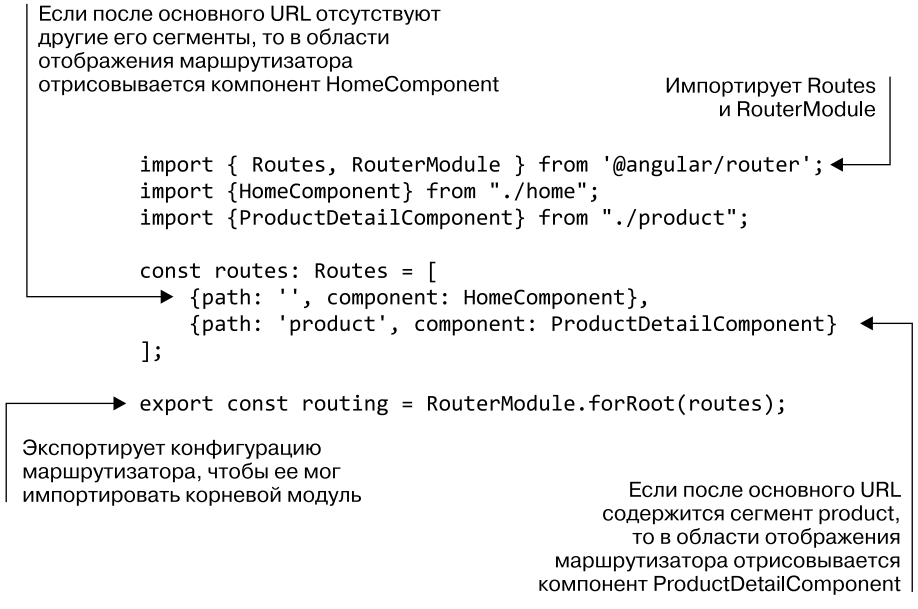
Код компонента `ProductDetailComponent` выглядит похоже, но вместо красного цвета фона в нем используется голубой (листинг 3.2).

Листинг 3.2. Код компонента ProductDetailComponent

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

@Component({
    selector: 'product',
    template: '<h1 class="product">Product Details Component</h1>',
    styles: ['.product {background: cyan}']})
export class ProductDetailComponent {}
```

Сконфигурируем маршруты в отдельном файле, который называется `app.routing.ts` (листинг 3.3).

Листинг 3.3. Содержимое файла app.routing.ts



Элемент `HomeComponent` соотносится с путем, содержащим пустую строку, что неявно делает его маршрутом по умолчанию.

Тип `Routes` представляет собой коллекцию объектов, чьи свойства объявлены в интерфейсе `Route`, как показано в следующем фрагменте кода:

```
export interface Route {
    path?: string;
    pathMatch?: string;
    component?: Type | string;
    redirectTo?: string;
    outlet?: string;
    canActivate?: any[];
    canActivateChild?: any[];
    canDeactivate?: any[];
    canLoad?: any[];
    data?: Data;
    resolve?: ResolveData;
    children?: Route[];
    loadChildren?: string;
}
```

Интерфейсы TypeScript описаны в приложении Б, но мы хотели бы напомнить: вопросительный знак, стоящий после свойства, подразумевает, что значение этого свойства задавать необязательно. Можно передать функциям `forRoot()` или

`forChild()` объект конфигурации, в котором заполнено лишь несколько свойств. В простом приложении вы будете использовать только два свойства класса `Route`: `path` и `component`.

Следующий шаг заключается в создании корневого компонента, который будет содержать ссылки для навигации между представлениями Home (Главная страница) и Product Details (Информация о продукте). Корневой компонент `AppComponent` расположится в файле `app.component.ts` (листинг 3.4).

Листинг 3.4. Содержимое файла app.component.ts

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

@Component({
    selector: 'app',
    template: `
      <a [routerLink]="['/']">Home</a>
      <a [routerLink]="['/product']">Product Details</a>
      <router-outlet></router-outlet>
    `
})
export class AppComponent {}
```

Создает ссылку, привязывающую routerLink к пустому пути

Создает ссылку, которая привязывает routerLink к пути /product

Тег <router-outlet> указывает область страницы, где маршрутизатор отрисует компоненты (по одному за раз)

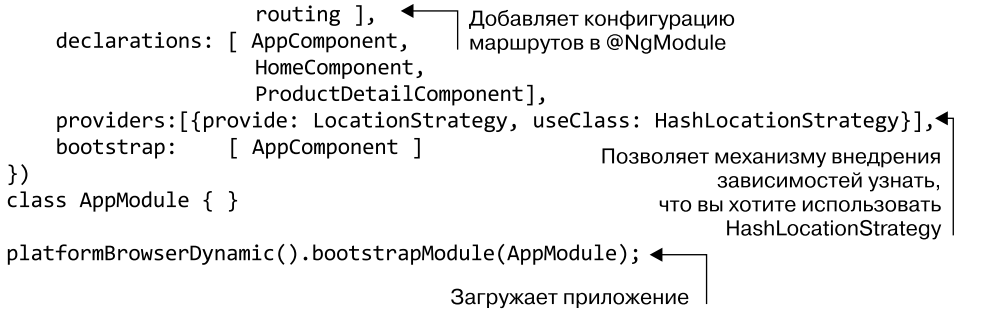
Обратите внимание на использование скобок в тегах `<a>`. Квадратные скобки, окружающие `routerLink`, указывают на привязку свойств, а скобки справа представляют массив с одним элементом (например, `['/']`). Мы покажем вам примеры массивов, содержащих два элемента и более, далее в этой главе. Второй якорный тег имеет свойство `routerLink`, привязанное к компоненту, сконфигурированному так, что на него указывает путь `/product path`. Соответствующие элементы будут отрисованы в области, помеченной тегом `<router-outlet>`, которая в данном приложении находится под якорными тегами.

Ни один компонент не знаком с конфигурацией маршрутизатора, поскольку за нее отвечает модуль. Объявим и загрузим корневой модуль. Для простоты реализуем оба эти действия в одном файле — `main.ts` (листинг 3.5).

Листинг 3.5. Содержимое файла main.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent }  from './components/app.component';
import {HomeComponent} from "./components/home";
import {ProductDetailComponent} from "./components/product";
import {LocationStrategy, HashLocationStrategy} from '@angular/common';
import {routing} from './components/app.routing';

@NgModule({
    imports:      [ BrowserModule,
```

Импортирует конфигурацию маршрутов



Свойство модуля `providers` представляет собой массив зарегистрированных провайдеров (в нашем примере показан только один) для внедрения зависимостей; эту тему мы рассмотрим в главе 4.

Сейчас же вам нужно знать лишь одно: несмотря на то, что стратегией навигации по умолчанию является `PathLocationStrategy`, Angular должен использовать для маршрутизации класс `HashLocationStrategy` (обратите внимание на символ решетки в URL на рис. 3.4).

ПРИМЕЧАНИЕ

Angular удаляет хвостовые слэши из всех URL. Вы можете увидеть, как выглядят URL для этих маршрутов, на рис. 3.3 и 3.4. Компоненты-потомки могут иметь собственную конфигурацию маршрутов; данный вопрос мы рассмотрим позже в настоящей главе.

Запуск примеров приложений из этой книги

Как правило, код, поставляемый с каждой главой, представляет собой несколько приложений. Чтобы запустить определенное приложение, нужно изменить одну строку в файле конфигурации SystemJS — указать имя основного сценария, который вы хотите запустить.

Для запуска этого приложения с помощью кода, поставляемого вместе с данной книгой, убедитесь, что основной сценарий, выполняющий изначальную загрузку вашего корневого модуля, верно соотнесен в файле `systemjs.config.js`.

Например, именно так можно указать, что основной сценарий находится в файле `main-param.ts`:

```
packages: {
  'app': {main: 'main-param', defaultExtension: 'ts'}
}
```

Данное выражение верно и для других приложений этой и других глав.

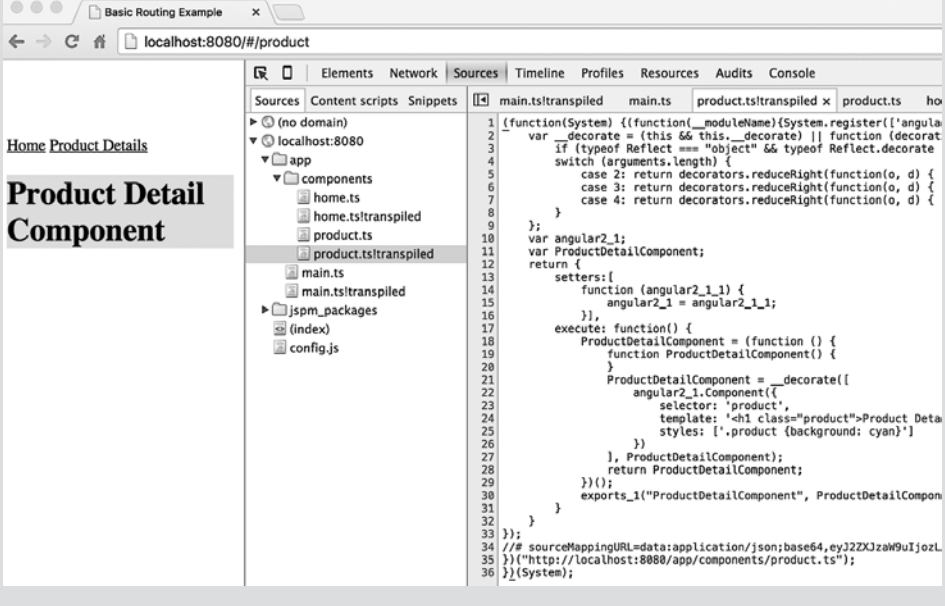
Основной сценарий данного приложения находится в файле `main.ts` в каталоге `samples`. Для запуска приложения убедитесь, что в файле `systemjs.config.js`

указан файл `main.ts` в пакете `app`, а затем запустите live-сервер из корневого каталога проекта.

SystemJS и динамическая компиляция

TypeScript предлагает элегантный декларативный синтаксис для реализации множества особенностей Angular, включая маршрутизацию. Мы используем SystemJS в наших примерах кода, а компиляция кода TypeScript в код JavaScript выполняется динамически, когда код приложения загружается в браузер.

Но если приложение работает не так, как вы того ожидаете? Открыв панель Developer Tools (Инструменты разработчика) в своем браузере, вы увидите, что для каждого файла с расширением `.ts` имеется соответствующий скомпилированный файл с расширением `.ts!transpiled`. Он содержит скомпилированную версию кода, которая может быть полезна, если вам нужно увидеть реальный код JavaScript, запускающийся в браузере. На следующем рисунке показана панель Developer Tools (Инструменты разработчика) браузера Chrome, отображающая исходный код файла `product.ts!transpiled`.



ПРИМЕЧАНИЕ

В Angular имеется класс Location, позволяющий выполнять навигацию для абсолютного URL, вызывая методы go(), forward() и back() (а также некоторые другие методы). Этот класс следует применять только в том случае, если нужно взаимодействовать с URL, не затрагивая маршрутизатор Angular. Вы увидите пример использования класса Location в главе 9, где будете писать сценарии для модульного тестирования.

3.1.3. Навигация по маршрутам с помощью метода navigate()

В примере кода приложения basic_routing, показанного в предыдущем разделе, вы настроили навигацию с помощью свойств `routerLink`, расположенных в якорных тегах HTML. Но что если нужно настроить навигацию программно, не требуя от пользователя нажимать ссылки? Модифицируем этот пример кода так, чтобы навигация совершалась с использованием метода `navigate()`. Вы добавите кнопку, после нажатия которой тоже будет выполняться переход к компоненту `ProductDetailComponent`, но в этот раз мы не будем пользоваться якорями HTML.

В листинге 3.6 (файл `main-navigate.ts`) будет вызываться метод `navigate()` экземпляра класса `Router`, который будет внедрен в компонент `RootComponent` через конструктор. Для простоты мы поместим модули, объявление маршрутов, начальную загрузку и компонент `AppComponent` в один файл, но в реальных проектах следует держать их отдельно друг от друга, как мы делали это в предыдущих разделах.

Листинг 3.6. Содержимое файла main-navigate.ts

```
import {Component} from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import {LocationStrategy, HashLocationStrategy} from '@angular/common';
import { Router, Routes, RouterModule } from '@angular/router';
import {HomeComponent} from "./components/home";
import {ProductDetailComponent} from "./components/product";

const routes: Routes = [
    {path: '', component: HomeComponent},
    {path: 'product', component: ProductDetailComponent}
];

@Component({
    selector: 'app',
    template: `
        <a [routerLink]="['/']">Home</a>
        <a [routerLink]="['/product']">Product Details!!!</a>
        <input type="button" value="Product Details"
               (click)="navigateToProductDetail()" />
        <router-outlet></router-outlet>
    `
})
class AppComponent {

    constructor(private _router: Router){}

    navigateToProductDetail(){

        this._router.navigate(["/product"]);
    }
}
```

Нажатие этой кнопки вызовет метод navigateToProductDetail()

Angular внедрит экземпляр класса Router в переменную router

Программно переходит к сконфигурированному маршруту продукта

```
@NgModule({
    imports: [ BrowserModule, RouterModule.forRoot(routes)],
    declarations: [ AppComponent, HomeComponent, ProductDetailComponent],
    providers:[{provide: LocationStrategy, useClass: HashLocationStrategy}],
    bootstrap: [ AppComponent ]
})
class AppModule { }

platformBrowserDynamic().bootstrapModule(AppModule);
```

В данном примере для навигации по маршруту продукта применяется кнопка, но это можно сделать и программно, не требуя от пользователя совершения каких-либо действий. Просто при необходимости вызовите метод `navigate()` (или `navigateByUrl()`) из кода приложения. Вы увидите еще один пример использования этого API в главе 9, где мы объясним, как выполнять модульное тестирование маршрутизатора.

СОВЕТ

Наличие ссылки на экземпляр класса Router позволяет проверить, является ли заданный маршрут активным, путем вызова метода isRouteActive().

Обработка ошибок 404

Если пользователь введет несуществующий URL в вашем приложении, то маршрутизатор не сможет найти соответствующий маршрут и выведет сообщение об ошибке в консоли браузера, оставив пользователя недоумевать, почему навигация не работает. Следует задуматься над созданием компонента приложения, который будет отображаться, когда приложение не может найти нужный пользователю элемент.

Например, можно создать компонент с именем `_404Component` и сконфигурировать его так, чтобы он использовал универсальный путь `**`:

```
[
  {path: '',        component: HomeComponent},
  {path: 'product', component: ProductDetailComponent},
  {path: '**', component: _404Component}
])
```

Теперь, когда маршрутизатор не сможет найти компонент, соответствующий заданному URL, он отрисует компонент `_404Component`. Вы можете посмотреть, как это работает, запустив приложение `main-with-404.ts`, которое поставляется с данной книгой. Просто введите в браузер несуществующий URL, например, http://localhost:8080/#/wrong.

Такая конфигурация должна замыкать массив маршрутов. Маршрутизатор всегда считает подобные маршруты соответствующими запросам пользователя, поэтому все маршруты, которые следуют после универсального маршрута, он рассматривать не будет.

3.2. Передача данных маршрутам

Простое приложение для маршрутизации показало, как можно отображать разные компоненты в заранее определенной области окна, но зачастую нужно не только отобразить элемент, но еще и передать ему какие-то данные. Например, при переходе от представления Home (Главная страница) к представлению Product Details (Информация о продукте) нужно передать идентификатор продукта компоненту, который представляет собой место назначения, например, `ProductDetailComponent`.

Компонент может получать передаваемые параметры в качестве аргументов конструктора, имеющего тип `ActivatedRoute`. Помимо переданных параметров, `ActivatedRoute` хранит сегмент URL маршрута, который представляет собой область отображения. Мы покажем вам, как извлекать параметры маршрута из объекта `ActivatedRoute`, далее в этом разделе.

3.2.1. Извлечение параметров из объекта ActivatedRoute

Когда пользователь переходит по маршруту `Product Details`, вам нужно передать этому маршруту его идентификатор, чтобы отобразить подробную информацию об определенном продукте. Изменим код приложения из предыдущего раздела так, чтобы компонент `RootComponent` мог передавать идентификатор продукта компоненту `ProductDetailComponent`.

Новая версия этого компонента будет называться `ProductDetailComponentParam`, и Angular внедрит в него объект типа `ActivatedRoute` (листинг 3.7). Он будет содержать информацию о компоненте, загруженном в область отображения.

Листинг 3.7. Содержимое параметра ProductDetailComponentParam

```
import {Component} from '@angular/core';
import {ActivatedRoute} from '@angular/router';

@Component({
  selector: 'product',
  template: `<h1 class="product">Product Details for Product:
  ➥ {{productID}}</h1>`,
  styles: ['.product {background: cyan}']
})
export class ProductDetailComponentParam {
  productID: string;

  constructor(route: ActivatedRoute) {
    this.productID = route.snapshot.params['id'];
  }
}
```

Отображает полученный идентификатор продукта с помощью привязки

Конструктор этого компонента просит Angular внедрить объект типа ActivatedRoute

Получает значение параметра с именем id и присваивает его переменной класса productID, которая используется в шаблоне, благодаря привязке

Объект класса `ActivatedRoute` будет содержать все параметры, передаваемые компонентам. Вам лишь нужно объявить аргумент конструктора, указав его тип, и Angular узнает, как создавать и внедрять этот объект. Более подробно тему внедрения зависимостей мы рассмотрим в главе 4.

В листинге 3.8 вы измените конфигурацию маршрута `product` и `routerLink` для гарантии того, что значение идентификатора продукта будет передано в компонент `ProductDetailComponentParam`, если пользователь выберет пойти по этому маршруту. Новая версия приложения называется `main-param.ts`.

Листинг 3.8. Содержимое файла main-param.ts

```
import {Component} from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import {LocationStrategy, HashLocationStrategy} from '@angular/common';
import { Routes, RouterModule } from '@angular/router';
import {HomeComponent} from "./components/home";
import {ProductDetailComponentParam} from "./components/product-param";

const routes: Routes = [
    {path: '', component: HomeComponent},
    {path: 'product/:id', component: ProductDetailComponentParam}
];

@Component({
    selector: 'app',
    template: `
        <a [routerLink]="['/']">Home</a>
        <a [routerLink]="['/product', 1234]">Product Details</a>
        <router-outlet></router-outlet>
    `
})
class AppComponent {}

@NgModule({
    imports:      [ BrowserModule, RouterModule.forRoot(routes)],
    declarations: [ AppComponent, HomeComponent, ProductDetailComponentParam],
    providers:[{provide: LocationStrategy, useClass: HashLocationStrategy}],
    bootstrap:    [ AppComponent ]
})
class AppModule { }

platformBrowserDynamic().bootstrapModule(AppModule);
```

Свойство path имеет дополнительный сегмент URL, /:id

В этот раз в routerLink передается два элемента массива: путь, с которого начинается маршрут, и число, представляющее собой идентификатор продукта

Свойство `routerLink` ссылки `Product Details` инициализировано с помощью двухэлементного массива. Элементы массива являются составными частями пути, указанного в конфигурации маршрута, переданной в метод `RouterModule.forRoot()`. Первый элемент массива представляет собой статическую часть пути маршрута: `product`. Второй элемент представляет собой переменную часть пути: `/:id`.

Для простоты мы жестко закодируем значение идентификатора `1234`, но если класс `RootComponent` имеет переменную `productID`, указывающую на соответствующий объект, то вы можете использовать конструкцию `{ productID}` вместо значения `1234`. Для маршрута Product Details (Информация о продукте) Angular создаст сегмент URL `/product/1234`. На рис. 3.5 показано, как представление Product Details

(Информация о продукте) будет отрисовано в браузере. Обратите внимание на URL: маршрутизатор заменил путь `product/:id` на путь `/product/1234`.

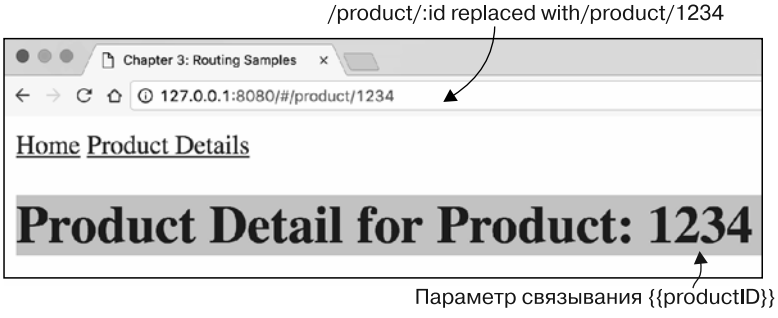


Рис. 3.5. Маршрут Product Details получил идентификатор продукта 1234

Рассмотрим действия, которые выполнил Angular для отрисовки основной страницы приложения.

1. Проверил содержимое каждого элемента `routerLink`, чтобы найти соответствующие конфигурации маршрута.
2. Проанализировал доступные URL и заменил имена параметров реальными значениями там, где они были указаны.
3. Создал теги `<a href="">`, которые понимает браузер.

На рис. 3.6 показан снимок главной страницы и открытой панели Developer Tools (Инструменты разработчика) браузера Chrome. Поскольку свойство `path` сконфигурированного маршрута `Home` содержит пустую строку, Angular ничего не добавляет в основной URL страницы. Но якорь под ссылкой `Product Details` уже был сконвертирован в обычный тег HTML. Когда пользователь нажмет эту ссылку, маршрутизатор прикрепит к базовому URL символ решетки и конструкцию `/product/1234`. Абсолютный URL представления Product Details (Информация о продукте) будет выглядеть так: http://localhost:8080/#/product/1234.

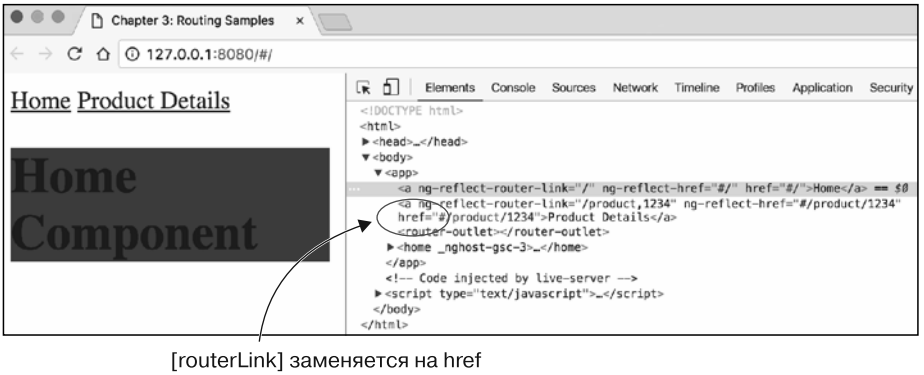


Рис. 3.6. Якорный тег для Product Details готов

3.2.2. Передача статических данных маршруту

Родительские компоненты обычно передают данные своим потомкам, но Angular также предлагает механизм передачи произвольных данных элементам в момент конфигурирования маршрута. Например, за исключением динамических данных, таких как идентификаторы продуктов, вам может понадобиться передать флаг, указывающий на то, что приложение запущено в производственной среде. Это можно сделать с помощью свойства `data` объекта конфигурации вашего маршрута.

Маршрут для представления, содержащего подробную информацию о продукте, может быть сконфигурирован следующим образом:

```
{path: 'product/:id', component: ProductDetailComponentParam , data:
➥ [{isProd: true}]}
```

Свойство `data` может содержать массив произвольных пар «ключ — значение». Когда маршрутизатор открывает `ProductDetailComponentParam`, данные будут находиться в свойстве `data ActivatedRoute.snapshot`:

```
export class ProductDetailComponentParam {
  productID: string;
  isProdEnvironment: string;
  constructor(route: ActivatedRoute) {
    this.productID = route.snapshot.params['id'];
    this.isProdEnvironment = route.snapshot.data[0]['isProd'];
    console.log("this.isProdEnvironment = " + this.isProdEnvironment);
  }
}
```

Передача данных маршруту с помощью свойства `data` не является альтернативой конфигурированию параметров в свойстве `path`, как, например, здесь: `'product/:id'`. Но такая функциональность может пригодиться, если вам нужно передать какие-то данные маршруту на этапе конфигурирования, например, информацию о том, в какой среде работает программа. Приложение, которое реализует эту функциональность, находится в файле `main-param-data.ts`.

3.3. Маршруты-потомки

Angular-приложение представляет собой дерево компонентов, связанных отношениями предок — потомок. Каждый компонент инкапсулирован, и вы сами решаете, какие элементы сделать доступными остальным сценариям приложения, а какие — закрытыми в рамках компонента. Любой элемент может иметь собственные стили, которые не будут смешиваться со стилями предка. Компонент также может иметь собственные инъекторы для зависимостей. Компонент-потомок может иметь собственные маршруты, но все маршруты конфигурируются за пределами элементов.

В предыдущем разделе вы сконфигурировали маршруты, чтобы показывать содержимое либо `HomeComponent`, либо `ProductDetailComponent` в `router-outlet` компонента `AppComponent`.

Допустим, вам нужно дать возможность `ProductDetailComponent` (потомку) показывать либо описание продукта, либо информацию о продавце. Это значит, что вам нужно добавить конфигурацию маршрутов-потомков в компонент `ProductDetailComponent`. Здесь следует использовать свойство `children` интерфейса `Route`:

```
[ {path: '',            component: HomeComponent},
  {path: 'product/:id', component: ProductDetailComponent,
    children: [
      {path: '', component: ProductDescriptionComponent},
      {path: 'seller/:id', component: SellerInfoComponent}
    ]}
]
```

На рис. 3.7 показано, как приложение будет выглядеть после того, как пользователь нажмет ссылку `Product Details` в корневом компоненте, который отрисует элемент `ProductDetailComponent` (потомка), показывая `ProductDescription`. Это маршрут по умолчанию для потомка, поскольку его свойство `path` содержит пустую строку.

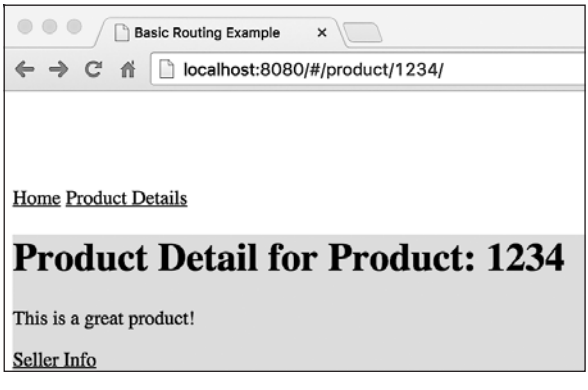


Рис. 3.7. Маршрут Product Description

На рис. 3.8 показан внешний вид приложения после того, как пользователь нажмет ссылку `Product Details`, а затем ссылку `Seller Info`.

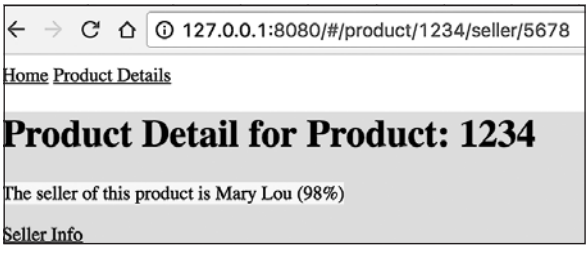


Рис. 3.8. Маршрут-потомок отрисовывает SellerInfo

ПРИМЕЧАНИЕ

На самом деле информация о продавце показывается на желтом фоне. Мы сделали фон таким для того, чтобы позже в данной главе обсудить задание стилей компонентов.

Для реализации представлений, показанных на рис. 3.7 и 3.8, нужно модифицировать компонент `ProductDetailComponent` так, чтобы у него было два потомка, а также собственный `<router-outlet>`. На рис. 3.9 показана иерархия компонентов, которые вы реализуете.

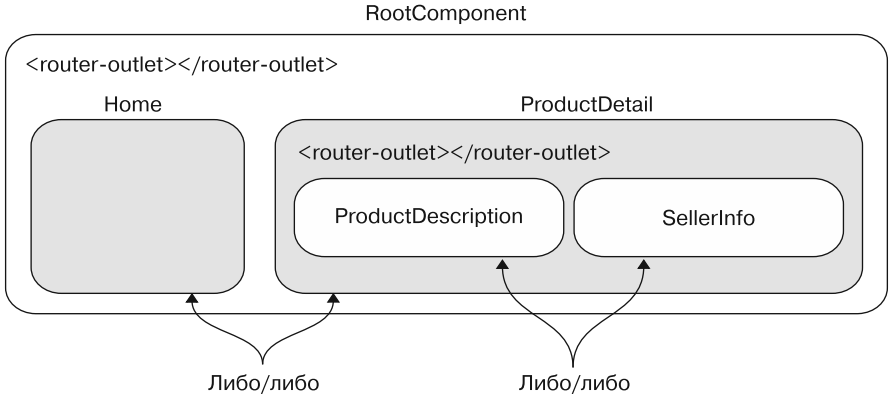


Рис. 3.9. Иерархия маршрутов в приложении basic_routing

Весь код этой главы, включая маршруты-потомки, расположен в файле `main-child.ts`, показанном далее (листинг 3.9).

Листинг 3.9. Содержимое файла main-child.ts

```
import {Component} from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import {LocationStrategy, HashLocationStrategy} from '@angular/common';
import { Routes, RouterModule } from '@angular/router';
import {HomeComponent} from "./components/home";
import {ProductDetailComponent} from './components/product-child';
import {ProductDescriptionComponent} from './components/product-description';
import {SellerInfoComponent} from './components/seller';

const routes: Routes = [
    {path: '',            component: HomeComponent},
    {path: 'product/:id', component: ProductDetailComponent,
        children: [
          {path: '',           component: ProductDescriptionComponent},
          {path: 'seller/:id', component: SellerInfoComponent}
        ]}
];
```

```
@Component({
    selector: 'app',
    template: `
        <a [routerLink]="['/']">Home</a>
        <a [routerLink]="['/product', 1234]">Product Details</a>
        <router-outlet></router-outlet>
    `
})
class AppComponent {}

@NgModule({
    imports:      [ BrowserModule, RouterModule.forRoot(routes)],
    declarations: [ AppComponent, HomeComponent, ProductDetailComponent,
                    ProductDescriptionComponent, SellerInfoComponent],
    providers:[{provide: LocationStrategy, useClass: HashLocationStrategy}],
    bootstrap: [ AppComponent ]
})
class AppModule { }
platformBrowserDynamic().bootstrapModule(AppModule);
```

Еще раз взглянем на URL, показанный на рис. 3.8. Когда пользователь нажимает ссылку, к URL добавляется сегмент `product/1234`. Маршрутизатор находит соответствие этому пути в объекте конфигурации и отрисовывает в области отображения компонент `ProductDetailComponent`.

Новая версия компонента `ProductDetailComponent` (`product-child.ts`) имеет собственную область отображения, где он может отобразить либо компонент `ProductDescriptionComponent` (по умолчанию), либо компонент `SellerInfoComponent` (листинг 3.10).

Листинг 3.10. Новая версия компонента ProductDetailComponent

```
import {Component} from '@angular/core';
import {ActivatedRoute} from '@angular/router';

@Component({
    selector: 'product',
    styles: ['.product {background: cyan}'],
    template: `
      <div class="product">
        <h1>Product Details for Product: {{productID}}</h1>
        <router-outlet></router-outlet>
        <p><a [routerLink]="['./seller', 5678]">Seller Info</a></p>
      </div>
    `
})
export class ProductDetailComponent {
  productID: string;

  constructor(route: ActivatedRoute) {
    this.productID = route.snapshot.params['id'];
  }
}
```

Компонент ProductDetailComponent имеет собственный элемент router-outlet, который используется для отрисовки компонентов-потомков (по одному за раз)

Когда пользователь нажимает эту ссылку, Angular добавляет сегмент /seller/5678 в существующий URL и отрисовывает компонент SellerInfoComponent

ПРИМЕЧАНИЕ

Элементы-потомки не нужно импортировать. Компоненты ProductDescriptionComponent и SellerInfoComponent не упоминаются явно в шаблоне компонента ProductDetailComponent, и вам не нужно указывать их в свойстве directives. Они включены в AppModule.

Конфигурация маршрута для компонента `SellerInfoComponent` говорит о том, что он ожидает получить в качестве параметра идентификатор продавца. Вы будете передавать жестко закодированное значение `5678` в качестве идентификатора продавца.

Когда пользователь нажимает ссылку Seller Info (Информация о продавце), URL будет содержать сегмент `product/1234/seller/5678` (см. рис. 3.8). Маршрутизатор найдет соответствие в объекте конфигурации и отобразит компонент `SellerInfoComponent`.

ПРИМЕЧАНИЕ

Эта версия компонента ProductDetailComponent может переходить только по одной ссылке к информации о продавце. Для перехода от информации о продавце к маршруту /product пользователь может просто нажать кнопку Back (Назад) в своем браузере.

Компонент `ProductDescriptionComponent` выглядит тривиально (листинг 3.11).

Листинг 3.11. Содержимое компонента ProductDescriptionComponent

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

@Component({
    selector: 'product-description',
    template: '<p>This is a great product!</p>'
})
export class ProductDescriptionComponent {}
```

Поскольку компонент `SellerInfoComponent` ожидает получить идентификатор продавца, его конструктор должен иметь аргумент типа `ActivatedRoute` (листинг 3.12); вы уже делали это для компонента `ProductDetailComponent`.

Листинг 3.12. Содержимое компонента SellerInfoComponent

```
import {Component} from '@angular/core';
import {ActivatedRoute} from '@angular/router';

@Component({
    selector: 'seller',
    template: 'The seller of this product is Mary Lou (98%)',
    styles: [':host {background: yellow}']
})
export class SellerInfoComponent {
  sellerID: string;
  constructor(route: ActivatedRoute){
    this.sellerID = route.snapshot.params['id'];
    console.log(`The SellerInfoComponent got the seller id ${this.sellerID}`);
  }
}
```

Вы применяете псевдокласс `:host`, чтобы отобразить содержимое этого компонента на желтом фоне. Сейчас кратко рассмотрим Shadow DOM.

Селектор псевдокласса `:host` может быть использован вместе с элементами, созданными с помощью Shadow DOM, предоставляющим более качественную инкапсуляцию для компонентов (см. врезку «Поддержка Shadow DOM в Angular» ниже). Несмотря на то, что еще не все браузеры поддерживают Shadow DOM, Angular эмулирует его по умолчанию и создает теневой корневой элемент. Элемент HTML, связанный с этим теневым корневым элементом, называется теневым хостом.

В листинге 3.12 вы используете `:host` для того, чтобы сделать желтый фон для компонента `SellerInfoComponent`, который служит теневым хостом. Стили элементов Shadow DOM не объединяются со стилями глобального DOM, а идентификаторы тегов HTML вашего компонента не пересекаются с идентификаторами DOM.

Глубокое связывание

Глубокое связывание позволяет создавать ссылки на определенное содержимое внутри веб-страницы вместо того, чтобы создавать ссылку на всю страницу. В простых приложениях с маршрутизацией вы уже видели примеры глубокого связывания.

- URL http://localhost:8080/#/product/1234 указывает не просто на страницу `Product Details`, но на отдельное представление, содержащее информацию о продукте с идентификатором `1234`.
- URL http://localhost:8080/#/product/1234/seller/5678 указывает на еще более глубокую страницу. Он показывает информацию о продавце с идентификатором `5678`, который продает продукт с идентификатором `1234`.

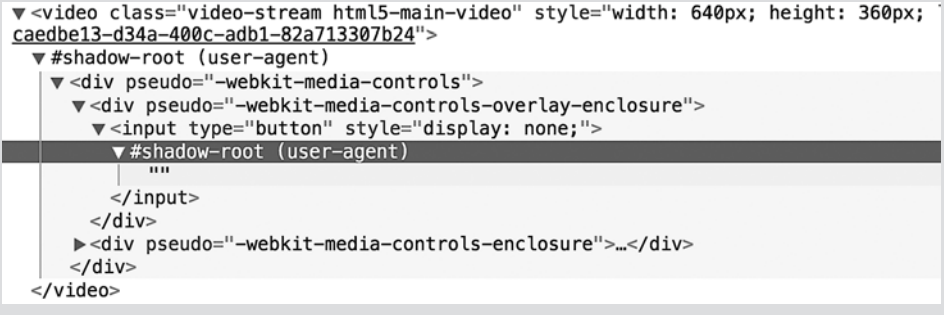
Вы можете увидеть глубокое связывание в действии, скопировав ссылку http://localhost:8080/#/product/1234/seller/5678 из приложения, запущенного в браузере Chrome, и вставив ее в браузеры Firefox или Safari.

Поддержка Shadow DOM в Angular

Shadow DOM — часть стандарта Web Components. Каждая веб-страница представлена деревом объектов DOM, но Shadow DOM позволяет инкапсулировать поддерево элементов HTML для того, чтобы создать границы между компонентами. Такое поддерево отрисовывается как часть документа HTML, но его элементы не прикреплены к основному дереву DOM. Другими словами, Shadow DOM размещает стену между содержимым DOM и внутренностями компонента HTML.

В том случае когда вы добавляете пользовательский тег на веб-страницу, он содержит фрагмент HTML, а благодаря Shadow DOM этот фрагмент виден только в рамках одного компонента, он не объединяется с DOM веб-страницы. С помощью Shadow DOM стили CSS пользовательского компонента не объединяются с основным CSS модели DOM, что предотвращает возможные конфликты при отрисовке стилей.

Откройте любой видеоролик на YouTube в браузере Chrome, нативно поддерживающем Shadow DOM. На момент написания этой книги видеопроигрыватель представлен тегом `video`, который вы можете найти, открыв панель Developer Tools (Инструменты разработчика) и взглянув на содержимое вкладки Elements (Элементы), как показано на следующем рисунке.



Несмотря на то, что видеопроигрыватель состоит из области содержимого и панели инструментов, включающей десяток кнопок (кнопка Play, ползунок звука и т. д.), все они инкапсулированы внутри теневого корневого элемента. С точки зрения основного DOM, эта страница содержит «деталь Lego» `<video>`. Чтобы заглянуть внутрь данного тега, нужно выбрать действие Show User Agent Shadow DOM в настройках панели Developer Tools (Инструменты разработчика).

В компонентах Angular вы указываете разметку HTML в свойствах `template` или `templateUrl` аннотации `@Component`. Если браузер нативно поддерживает Shadow DOM или вы указали, что Angular должен эмулировать его, код HTML-компонента не объединяется с глобальным объектом DOM веб-страницы. В Angular можно дать команду использовать режим Shadow DOM, установив свойству `encapsulation` аннотации `@Component` одно из следующих значений:

- `ViewEncapsulation.Emulated` — эмулирует инкапсуляцию Shadow DOM (значение по умолчанию). Указывает Angular сгенерировать уникальные атрибуты для стилей элемента и не объединять его стили со стилями модели DOM веб-страницы. Например, если вы откроете панель Developer Tools (Инструменты разработчика) в браузере Chrome при навигации по компоненту `SellerInfoComponent`, то разметка HTML этого компонента будет выглядеть так:

```
<head>
...
  <style>[_nghost-yls-7] {background: yellow;}</style>
</head>
...
<seller _nghost-yls-7="" _ngcontent-yls-6="">
  <p _ngcontent-yls-7=""></p>
  The seller of this product is Mary Lou (98% positive feedback)
</seller>
```

- `ViewEncapsulation.Native` — использует Shadow DOM, который нативно поддерживается браузером. HTML и стили не объединяются с моделью DOM веб-страницы.

Этот вариант следует применять только в том случае, если вы уверены, что браузер пользователя поддерживает Shadow DOM; в противном случае генерируется ошибка. В таком режиме стили компонента `SellerInfoComponent` не будут добавлены в раздел `<head>` вашей страницы, но все стили компонента и его предков будут инкапсулированы внутри компонента.

- `ViewEncapsulation.None` — не использует инкапсуляцию Shadow DOM. Вся разметка и стили будут интегрированы в глобальную модель DOM веб-страницы. Селектор `:host` не станет работать в этом режиме, поскольку не будет никакого теневого хоста. Вы все еще можете задать стиль для компонента `SellerInfoComponent`, ссылаясь на него с помощью его селектора:

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

@Component({
    selector: 'seller',
    template: 'The seller of this product is Mary Lou (98%)',
    styles: ['seller {background: yellow}'],
    encapsulation: ViewEncapsulation.None
})
export class SellerInfoComponent {}
```

Angular не будет генерировать дополнительные атрибуты стиля и добавит следующую строку в раздел `<head>` вашей страницы:

```
<style>seller {background: yellow}</style>
```

В подразделе 6.2.3 вы увидите, как `ViewEncapsulation` влияет на отрисовку пользовательского интерфейса как с помощью Shadow DOM, так и без него.

3.4. Граничные маршруты

Теперь, когда вы умеете настраивать базовую навигацию с помощью маршрутизатора, рассмотрим несколько сценариев, которые требуют выполнять валидацию, чтобы решить, может ли пользователь (или программа) перейти по маршруту:

- открытие маршрута только в том случае, если пользователь аутентифицирован и авторизован;

- отображение составной формы, которая состоит из нескольких компонентов, и пользователю можно переходить на следующий раздел формы только в том случае, если данные, введенные в этом разделе, корректны;
- напоминание пользователю о несохраненных изменениях, если он пробует уйти с маршрута.

Маршрутизатор имеет привязки, которые позволяют контролировать навигацию по маршруту и при уходе с него. Можно использовать эти привязки для реализации любых вышеописанных сценариев, чтобы *оградить* маршруты.

ПРИМЕЧАНИЕ

Angular содержит несколько привязок жизненного цикла компонентов, позволяющих обрабатывать важные события в жизни элемента. Мы рассмотрим их в главе 6.

В разделе 3.1 мы говорили, что тип `Routes` представляет собой массив элементов, который соответствует интерфейсу `Route`, показанному ниже:

```
export interface Route {
    path?: string;
    pathMatch?: string;
    component?: Type | string;
    redirectTo?: string;
    outlet?: string;
    canActivate?: any[];
    canActivateChild?: any[];
    canDeactivate?: any[];
    canLoad?: any[];
    data?: Data;
    resolve?: ResolveData;
    children?: Route[];
    loadChildren?: string;
}
```

При конфигурации предыдущих маршрутов вы использовали три свойства из этого интерфейса: `path`, `component` и `data`. Теперь познакомимся со свойствами `canActivate` и `canDeactivate`, которые позволяют связывать маршруты и граничные операторы (охранников). По сути, вам нужно написать функцию для реализации логики проверки, которая вернет значение `true` или `false`, и присвоить ее одному из указанных свойств. Если вызов `canActivate()` граничного оператора вернет значение `true`, то пользователь сможет перейти по данному маршруту. Если вызов `canDeactivate()` вернет значение `true`, то пользователь сможет уйти с этого маршрута. Поскольку свойства `canActivate` и `canDeactivate` типа `Route` принимают в качестве своего значения массив, можно присвоить несколько функций (граничных операторов), когда нужно проверить более одного условия для того, чтобы разрешить или запретить навигацию по маршруту.

Обновим пример из подраздела 3.1.2 (имеющий ссылки `Home` и `Product Details`), для демонстрации того, как можно предохранить маршрут `product` от неавтори-

зованных пользователей. Чтобы этот пример оставался простым, вы не будете использовать настоящий сервис авторизации, статус авторизации будет генерироваться случайным образом.

Создайте граничный класс, реализующий интерфейс `CanActivate`, в котором объявлена всего одна функция: `canActivate()` (листинг 3.13). Она должна содержать логику приложения, которая возвращает значение `true` или `false`. Если функция возвращает значение `false` (пользователь не авторизован), то приложение не перейдет по маршруту и выведет сообщение об ошибке в консоли.

Листинг 3.13. Содержимое класса LoginGuard

```
import {CanActivate} from "@angular/router";
import {Injectable} from "@angular/core";

@Injectable()
export class LoginGuard implements CanActivate{
  canActivate() {
      return this.checkIfLoggedIn();
  }
private checkIfLoggedIn(): boolean{
      // Здесь будет находиться вызов сервиса авторизации
      // Сейчас же мы будем возвращать значение true или false,
      // выбранное случайным образом
      let loggedIn:boolean = Math.random() <0.5;
      if(!loggedIn){
          console.log("LoginGuard:
          ➥ The user is not logged in and can't navigate product details");
      }
      return loggedIn;
  }
}
```

Как видите, эта реализация функции `canActivate()` будет случайным образом возвращать значение `true` или `false`, эмулируя авторизацию пользователя.

Следующий шаг заключается в обновлении конфигурации маршрутизатора так, чтобы он использовал ваш граничный оператор. В следующем фрагменте кода показано, как можно сконфигурировать маршрутизацию приложения, которое имеет маршруты `Home` и `Product Details`. Последний защищен оператором `LoginGuard`:

```
[
  {path: '',        component: HomeComponent},
  {path: 'product', component: ProductDetailComponent, canActivate:
  ➥ [LoginGuard]}
]
```

Добавление одного или нескольких граничных операторов в массив, передаваемый свойству `canActivate`, автоматически вызовет всех операторов одного за другим. Если хотя бы один из них вернет значение `false`, то переход по маршруту будет запрещен.

Но кто же будет создавать объект класса `LoginGuard`? Angular сделает это за вас с помощью механизма внедрения зависимостей (он описан в главе 4), но вам нужно упомянуть этот класс в списке поставщиков, которые нужны для того, чтобы внедрение сработало. Добавьте имя `LoginGuard` к списку поставщиков в `@NgModule`:

```
@NgModule({
    imports:      [ BrowserModule, RouterModule.forRoot(routes)],
    declarations: [ AppComponent, HomeComponent, ProductDetailComponent],
    providers:[{provide: LocationStrategy, useClass: HashLocationStrategy}
                LoginGuard],
    bootstrap:    [ AppComponent ]
})
```

Ниже представлен полный код основного сценария приложения (файл `main-with-guard.ts`) (листинг 3.14).

Листинг 3.14. Содержимое файла main-with-guard.ts

```
import {Component} from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import {LocationStrategy, HashLocationStrategy} from '@angular/common';
import { Routes, RouterModule } from '@angular/router';
import {HomeComponent} from "./components/home";
import {ProductDetailComponent} from "./components/product";
import {LoginGuard} from "./guards/login.guard";

const routes: Routes = [
    {path: '',        component: HomeComponent},
    {path: 'product', component: ProductDetailComponent,
        canActivate:[LoginGuard]}
];

@Component({
    selector: 'app',
    template: `
        <a [routerLink]="['/']">Home</a>
        <a [routerLink]="['/product']">Product Details</a>
        <router-outlet></router-outlet>
    `
})
class AppComponent {}

@NgModule({
    imports:      [ BrowserModule, RouterModule.forRoot(routes)],
    declarations: [ AppComponent, HomeComponent, ProductDetailComponent],
    providers:[{provide: LocationStrategy, useClass: HashLocationStrategy},
                LoginGuard],
    bootstrap:    [ AppComponent ]
})
class AppModule { }
platformBrowserDynamic().bootstrapModule(AppModule);
```

Если вы запустите это приложение и нажмете ссылку `Product Details`, то приложение либо перейдет по данному маршруту, либо выведет сообщение об ошибке в консоли браузера в зависимости от сгенерированного случайным образом значения в `LoginGuard`. На рис. 3.10 показан снимок экрана, сделанный после того, как пользователь нажал ссылку `Product Details`, но граничный оператор `LoginGuard` решил, что тот не авторизован.

Рис. 3.10. Переход по ссылке Product Details охраняется

В данном примере вы реализовали метод `canActivate()`, не предоставляя ему никаких аргументов. Но этот метод можно использовать со следующей сигнатурой:

```
canActivate(destination: ActivatedRouteSnapshot, state: RouterStateSnapshot)
```

Значения `ActivatedRouteSnapshot` и `RouterStateSnapshot` Angular внедрит автоматически, и это может оказаться полезным, если вы хотите проанализировать текущее состояние маршрутизатора. Например, чтобы узнать имя маршрута, по которому пользователь пробует перейти, можно сделать так:

```
canActivate(destination: ActivatedRouteSnapshot, state: RouterStateSnapshot)
    {
    console.log(destination.component.name);
    ...
}
```

СОВЕТ

Если хотите подождать прихода неких асинхронных данных перед тем, как переходить по маршруту, то используйте свойство resolve при конфигурировании маршрута. В нем можно указать класс, который реализует интерфейс Resolve, имеющий функцию resolve(). Маршрутизатор не будет создавать экземпляр сконфигурированного компонента до тех пор, пока функция не вернет значение.

Реализация интерфейса `CanDeactivate`, который управляет процессом навигации по маршруту, выполняется аналогичным образом. Просто создайте граничный класс, реализующий метод `canDeactivate()`, например, так:

```
import {CanDeactivate, Router} from "@angular/router";
import {Injectable} from "@angular/core";

@Injectable()
export class UnsavedChangesGuard implements CanDeactivate{
    constructor(private _router:Router){}
    canDeactivate(){
        return window.confirm("You have unsaved changes.
        ➥ Still want to leave?");
    }
}
```

Не забудьте добавить свойство `canDeactivate` в конфигурацию маршрута, а также включить новый граничный оператор в список поставщиков вашего модуля:

```
@NgModule({
    imports:      [ BrowserModule, RouterModule.forRoot(routes)],
    declarations: [ AppComponent, HomeComponent, ProductDetailComponent],
    providers:[{provide: LocationStrategy, useClass: HashLocationStrategy},
                LoginGuard, UnsavedChangesGuard],
    bootstrap:    [ AppComponent ]
})
```

СОВЕТ

Более красивое отображение оповещения и диалогов подтверждения возможно с помощью компонента MdDialog, поставляющегося в библиотеке Material Design 2 (https://github.com/angular/material2).

Для получения более подробной информации о привязках жизненного цикла, применимых к навигации, обратитесь к разделу @angular/router документации к Angular API (https://angular.io/api). Мы рассмотрим жизненные циклы компонентов в главе 6.

3.5. Создание одностраничного приложения с несколькими областями отображения

Из предыдущего раздела вы узнали, что маршрут-потомок представлен URL, состоящим из сегментов-предков и сегментов-потомков. Ваше одностраничное приложение имеет один тег, `<router-outlet>`, где Angular отрисует компонент, сконфигурированный для предка или потомка. Теперь обсудим конфигурирование и отрисовку маршрутов одного уровня, что означает их отрисовку в отдельных областях отображения в одно и то же время. Рассмотрим несколько вариантов использования.

- Предположим, веб-клиент Gmail отображает список электронных писем во вкладке Inbox (Входящие), и вы хотите написать новое письмо. В правой части

окна будет отображено новое представление, и вы сможете переключиться между входящими письмами и черновиком нового письма, не закрывая ни одно из представлений.

- Представим одностраничное приложение — панель инструментов, в котором имеется несколько выделенных областей отображения, и в каждой из них может отображаться более одного компонента (но по одному за раз). В области отображения A вы можете показать портфолио акций либо как таблицу, либо как схему, а в области отображения B демонстрируется последние новости или реклама.
- Предположим, вам нужно добавить в одностраничное приложение чат, чтобы пользователь мог пообщаться с представителем службы поддержки, не закрывая текущий маршрут. По сути, вы хотите добавить независимый маршрут для чата, позволяя пользователю работать с обоими маршрутами одновременно, а также переключаться между ними.

В Angular вы можете реализовать любое количество подобных сценариев, создав помимо основной области отображения именованные вспомогательные области, которые отображаются вместе с основной.

Чтобы отделить отрисовку компонентов основных и вспомогательных маршрутов, нужно добавить еще один тег `<router-outlet>`, но эта область отображения должна иметь имя. Например, в следующем фрагменте кода определены основной `outlet` и `outlet` для чата:

```
<router-outlet></router-outlet>
<router-outlet name="chat"></router-outlet>
```

Добавим именованный маршрут для чата в наше приложение. На рис. 3.11 показаны два маршрута, открытые одновременно после того, как пользователь нажал ссылку `Home`, а затем и ссылку `Open Chat`. Слева показан отрисованный компонент `HomeComponent` в основной области отображения, а справа — `ChatComponent`, отрисованный в именованной области отображения. Нажатие ссылки `Close Chat` очистит содержимое именованной области отображения. (Мы добавили поля HTML `<input>` в компонент `HomeComponent` и `<textarea>` в компонент `ChatComponent`, чтобы было проще заметить, какой элемент имеет фокус, когда пользователь переключается между маршрутами `Home` и `Chat`.)

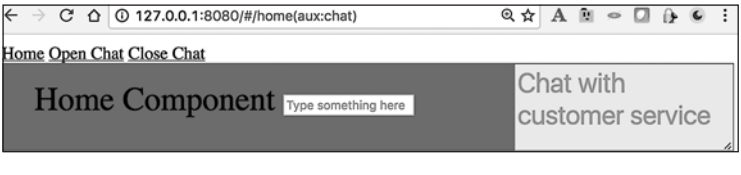


Рис. 3.11. Отрисовка представления чата с помощью вспомогательного маршрута

Обратите внимание на скобки в URL вспомогательного маршрута, http://localhost:8080/#home/(chat). Поскольку маршрут-потомок отделен от предка с помощью граничного слэша, вспомогательный маршрут представлен как сегмент

URL, расположенный в фигурных скобках. Этот URL говорит вам, что `Home` и `Chat` являются маршрутами одного уровня.

Код, реализующий данный фрагмент, располагается в файле `main_aux.ts`, он показан в листинге 3.15. Мы разместили все требуемые компоненты в одном файле для простоты. Компоненты `HomeComponent` и `ChatComponent` имеют встроенные стили, которые позволяют разместить их рядом друг с другом в окне. Компонент `HomeComponent` займет 70 % доступного пространства, а `ChatComponent` — остальные 30 %.

Листинг 3.15. Содержимое файла main_aux.ts

```
import {Component} from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import {LocationStrategy, HashLocationStrategy} from '@angular/common';
import { Routes, RouterModule } from '@angular/router';

@Component({
    selector: 'home',
    template: `<div class="home">Home Component
                <input type="text" placeholder="Type something here"/>
                ➥ </div>`,
    styles: [`.home {background: red; padding: 15px 0 0 30px; height: 80px;
    ➥ width:70%;
                    font-size: 30px; float:left; box-sizing:border-box;}`]})
export class HomeComponent {}

@Component({
    selector: 'chat',
    template: `<textarea placeholder="Chat with customer service"
                        class="chat"></textarea>`,
    styles: [`.chat {background: #eee; height: 80px;width:30%;
    ➥ font-size: 24px;
                    float:left; display:block; box-sizing:border-box;} `]})
export class ChatComponent {}

const routes: Routes = [
    {path: '', redirectTo: 'home', pathMatch: 'full'},

    {path: 'home', component: HomeComponent},
    {path: 'chat', component: ChatComponent, outlet:"aux"}
];

@Component({
    selector: 'app',
    template: `

      <a [routerLink]="['']">Home</a>
      <a [routerLink]="[{outlets: {primary: 'home',
      ➥ aux: 'chat'}}]">Open Chat</a>
```

Конфигурирует маршрут для компонента Home. Поскольку не указана конкретная область отображения, компонент будет отрисован в основной области отображения

Конфигурирует маршрут для компонента Chat, который будет отрисован в области отображения aux

Отрисовывает компонент Home в основной области отображения и компонент Chat в области отображения с именем aux

```
      <a [routerLink]="[{outlets:
      ➥ {aux: null}}]">Close Chat</a>
      <br/>

      <router-outlet></router-outlet>
      <router-outlet name="aux"></router-outlet>
    `
})
class AppComponent {}

@NgModule({
    imports: [ BrowserModule, RouterModule.forRoot(routes)],
    declarations: [ AppComponent, HomeComponent, ChatComponent],
    providers:[{provide: LocationStrategy, useClass: HashLocationStrategy}],
    bootstrap: [ AppComponent ]
})
class AppModule { }

platformBrowserDynamic().bootstrapModule(AppModule);
```

Для удаления именованной области отображения и ее содержимого задайте им значение null

Объявляет основную область отображения

Объявляет дополнительный именованный <router-outlet>. Здесь вы назвали его aux

ПРИМЕЧАНИЕ

Поскольку объявления классов не подняты («Поднятие» объясняется в приложении А), убедитесь, что объявляете компоненты до того, как используете их в routerLink.

Если хотите выполнить навигацию именованных областей отображения (или закрыть их) программно, используйте метод `Router.navigate()`, описанный в подразделе 3.1.3. Рассмотрим пример:

```
navigate([{outlets: {aux: 'chat'}}]);
```

Немного передохнем и вспомним, что мы уже узнали из этой главы:

- маршруты конфигурируются на уровне модулей;
- каждый маршрут имеет путь, соотнесенный с компонентом;
- область, где отрисовывается содержимое маршрута, определена расположением области `<router-outlet>` в шаблоне компонента;
- `routerLink` может быть использован при навигации по именованному маршруту;
- метод `navigate()` может применятся для навигации по именованному маршруту;
- если для маршрута требуется параметр, то нужно сконфигурировать его в свойстве `path` в конфигурации маршрута и передать его значение в `routerLink` или метод `navigate()`;
- если маршрут принимает параметр, то лежащий в его основе компонент должен иметь конструктор с аргументом типа `ActivatedRoute`;

- если компонент-потомок имеет собственную конфигурацию маршрутов, то он называется маршрутом-потомком и конфигурируется с помощью свойства `children`, определенного в интерфейсе `Route`;
- приложение может показывать больше одного маршрута одновременно, используя именованные маршруты.

Мы уже почти закончили рассматривать маршрутизатор. Нам осталось обсудить еще одну тему: реализацию ленивой загрузки компонентов для редко применяемых маршрутов. Это важный прием, который позволит минимизировать объем загружаемого кода для посадочной страницы приложения. Затем вы реализуете маршрутизацию для онлайн-аукциона.

3.6. Разбиение приложения на модули

Angular позволяет разбить приложение на несколько модулей, где каждый из них будет реализовывать определенную функциональность. В действительности каждый фрагмент кода, показанный в данной главе, уже содержит более одного модуля, например, `AppModule`, `BrowserModule` и `RouterModule`. Модуль `AppModule` является корневым модулем приложения, а `BrowserModule` и `RouterModule` — это модули функциональности. Обратите внимание на главное различие между ними: корневой модуль загружается предварительно, а модули функциональности импортируются, как показано в следующем фрагменте кода:

```
@NgModule({
    imports:[ BrowserModule,
              RouterModule.forRoot(routes)],
    ...
})
class AppModule { }
platformBrowserDynamic().bootstrapModule(AppModule);
```

Каждый модуль может предоставлять и скрывать определенную функциональность, все модули выполняются в одном контексте, так что могут иметь общие объекты. Модули `RootModule` и `BrowserModule` были созданы командой разработчиков Angular, но вы тоже можете разбить свое приложение на модули.

Для модулей функциональности декоратор `@NgModule` нужен для импортирования модуля `CommonModule` вместо `BrowserModule`. Возьмем приложение с двумя ссылками — `Home` и `Product Details` — и добавим еще одну: `Luxury Items`. Представьте, что предметы роскоши должны обрабатываться не так, как обычные продукты, и вам нужно вынести эту функциональность в отдельный модуль, который называется `LuxuryModule` и включает всего один компонент — `LuxuryComponent`. Модули функциональности и поддерживаемые ими элементы, сервисы и другие ресурсы рекомендуется размещать в отдельном каталоге. В нашем примере они будут находиться в каталоге с именем `luxury`.

Код модуля `LuxuryModule` находится в файле `luxury.module.ts`, показанном далее (листинг 3.16).

Листинг 3.16. Содержимое файла luxury.model.ts

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import {RouterModule} from '@angular/router';
import {LuxuryComponent} from "./luxury.component";

@NgModule({
    imports: [ CommonModule,
        RouterModule.forChild([
        {path: 'luxury', component: LuxuryComponent}
    ]) ],
    declarations: [ LuxuryComponent ]
})

export class LuxuryModule { }
```


Когда вы конфигурируете корневой модуль, используйте метод `forRoot`, для модулей функциональности применяйте метод `forChild()`.

Код компонента `LuxuryComponent` всего лишь отобразит текст Luxury Component на желтом (под цвет золота) фоне:

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

@Component({
    selector: 'luxury',
    template: `<h1 class="gold">Luxury Component</h1>`,
    styles: ['.gold {background: yellow}']
})
export class LuxuryComponent {}
```

Обратите внимание: вы экспортируете компонент `LuxuryComponent` для того, чтобы он стал доступным остальным членам корневого модуля. Код `AppComponent`, `AppModule` и функции предварительной загрузки находятся в файле `main-luxury.ts` (листинг 3.17).

Листинг 3.17. Содержимое файла main-luxury.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule, Component } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import {LocationStrategy, HashLocationStrategy} from '@angular/common';
import {RouterModule} from "@angular/router";
import {HomeComponent} from "./components/home";
import {ProductDetailComponent} from "./components/product";
import {LuxuryModule} from "./components/luxury/luxury.module";

@Component({
    selector: 'app',
    template: `
       <a [routerLink]="['/']">Home</a>
       <a [routerLink]="['/product']">Product Details</a>
```

```
        <a [routerLink]="['/luxury']">Luxury Items</a>
        <router-outlet></router-outlet>
    `
})
export class AppComponent {}

@NgModule({
    imports: [ BrowserModule,
               LuxuryModule,
               RouterModule.forRoot([
                   {path: '', component: HomeComponent},
                   {path: 'product', component: ProductDetailComponent}
                   ])
             ],
    declarations: [ AppComponent, HomeComponent, ProductDetailComponent],
    providers:[{provide: LocationStrategy, useClass: HashLocationStrategy}],
    bootstrap: [ AppComponent ]
})

class AppModule { }

platformBrowserDynamic().bootstrapModule(AppModule);
```

Добавляет ссылку на основное приложение для навигации по пути luxury

Объявляет модуль функциональности

Конфигурирует маршрут для корневого модуля

Обратите внимание на то, что корневой модуль не знает о содержимом модуля `LuxuryModule` и даже не упоминает `LuxuryComponent`.

Когда маршрутизатор анализирует конфигурацию маршрутов для корневых модулей и модулей функциональности, он корректно соотнесет путь `luxury` с компонентом `LuxuryComponent`, который был экспортирован модулем `LuxuryModule`. После запуска этого приложения и нажатия ссылки Luxury Items (Предметы роскоши) вы увидите окно, показанное на рис. 3.12.

В этом примере мы рассмотрели вопрос разбиения функциональности на модули. Если вы решите перестать продавать предметы роскоши, то вам понадобится лишь убрать ссылки на `LuxuryModule` из корневого модуля, а также одну ссылку из `AppComponent`. Такой рефакторинг выглядит довольно просто в сравнении с процессом удаления функциональности из монолитного одномодульного приложения.

Рис. 3.12. Отрисовка модуля LuxuryModule

Перемещение модуля функциональности из одного приложения в другое тоже становится проще. Не каждому приложению понадобится продавать предметы роскоши, но для многих приложений коммерческого портала может понадобиться, например, модуль оплаты, который можно использовать сразу для нескольких приложений, не прилагая особых усилий.

Это все хорошо, но помните: несмотря на то, что вы инкапсулировали какую-то функциональность в отдельный модуль, его код будет загружаться в момент запуска приложения. Вы действительно хотите загружать модуль `luxury` в браузер при запуске приложения? Обсудим этот вопрос далее.

3.7. «Ленивая» загрузка модулей

В крупных приложениях вы хотите минимизировать объем кода, который должен быть загружен для отрисовки посадочной страницы приложения. Чем меньше кода приложение загружает изначально, тем быстрее пользователь увидит его. Это особенно важно для мобильных приложений, когда они используются в тех местах, где соединение с Интернетом оставляет желать лучшего. Если ваше приложение имеет редко применяемые модули, то вы можете загружать их по требованию (или лениво).

Angular позволяет без труда разбить приложение на модули: один модуль будет корневым, а остальные — модулями функциональности. Последние могут быть загружены либо мгновенно, как было показано в предыдущем разделе, либо лениво.

После реализации функциональности для работы с предметами роскоши предположим, что вы узнали следующее: пользователи редко нажимают одноименную ссылку. Зачем тогда загружать код, который работает с предметами роскоши, при начальной загрузке приложения? Перепишем приложение так, чтобы этот код загружался по требованию.

В листинге 3.18 реализована ленивая загрузка модуля. Этот фрагмент выглядит практически так же, как и листинг 3.17, но мы внесем небольшое изменение в основной модуль, а также изменим способ экспортирования модуля `LuxuryModule`. Данный код находится в файле `main-luxury-lazy.ts`.

Листинг 3.18. Содержимое файла main-luxury-lazy.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule, Component }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import {LocationStrategy, HashLocationStrategy} from '@angular/common';
import {RouterModule} from "@angular/router";
import {HomeComponent} from "./components/home";
import {ProductDetailComponent} from "./components/product";

@Component({
    selector: 'app',
    template: `
        <a [routerLink]="['/']">Home</a>
        <a [routerLink]="['/product']">Product Details</a>
        <a [routerLink]="['/luxury']">Luxury Items</a>
        <router-outlet></router-outlet>
    `
})
export class AppComponent {}

@NgModule({
    imports: [ BrowserModule,
               RouterModule.forRoot([
                   {path: '', component: HomeComponent},
                   {path: 'product', component: ProductDetailComponent},
                   {path: 'luxury', loadChildren:
                   ➥ 'app/components/luxury/luxury.lazy.module'}
                   ])
```

```
            ],
    declarations: [ AppComponent, HomeComponent, ProductDetailComponent],
    providers:[{provide: LocationStrategy, useClass: HashLocationStrategy}],
    bootstrap:    [ AppComponent ]
})
class AppModule { }
platformBrowserDynamic().bootstrapModule(AppModule);
```

Обратите внимание на то, что сейчас вы не импортируете модуль `LuxuryModule` явно. Вы также изменяете конфигурацию для пути `luxury`, которая теперь выглядит так:

```
{path: 'luxury', loadChildren: 'app/components/luxury/luxury.lazy.module'}
```

Вместо того чтобы соотносить путь с компонентом, вы используете свойство `loadChildren`, предоставляя путь загружаемому модулю. Обратите внимание: значением `loadChildren` является не тип модуля, а строка. Корневой модуль не знает типа `LuxuryModule`, но, когда пользователь нажмет ссылку Luxury Items (Предметы роскоши), загрузчик модулей проанализирует эту строку и загрузит модуль `LuxuryModule` из файла `luxury.lazy.module.ts` (листинг 3.19), внешний вид которого отличается от версии, показанной в предыдущем разделе.

Листинг 3.19. Содержимое файла luxury.lazy.module.ts

```
import { NgModule }      from '@angular/core';
import { CommonModule } from '@angular/common';
import {RouterModule} from '@angular/router';
import {LuxuryComponent} from "./luxury.component";

@NgModule({
    imports:      [ CommonModule,
        RouterModule.forChild([
        {path: '', component: LuxuryComponent}
    ]) ],
    declarations: [ LuxuryComponent ]
})
export default class LuxuryModule { }
```

Здесь вы указываете, что пустой путь будет использоваться как маршрут по умолчанию. Поскольку этот модуль будет загружаться лениво и вы не объявляли тип `LuxuryModule` в корневом модуле, придется применить ключевое слово `default` при экспорте данного класса. Когда пользователь нажмет ссылку Luxury Items (Предметы роскоши) в корневом модуле, загрузчик загрузит содержимое файла `luxury.lazy.module.ts` и определит, что модуль `LuxuryModule` является точкой входа по умолчанию для сценария из этого файла.

Теперь, если вы запустите приложение main-luxury-lazy при открытой вкладке Network (Сеть) панели Developer Tools (Инструменты разработчика), то не увидите модуль `luxury` в списке загруженных файлов. Нажмите ссылку Luxury Items (Предметы роскоши), и вы увидите, как браузер делает дополнительный запрос на сервер для загрузки модуля `LuxuryModule` и компонента `LuxuryComponent`.

Для нашего крайне простого примера эти манипуляции снизили размер загружаемых данных всего на 1 Кбайт. Но при создании крупных приложений использование ленивой загрузки может снизить размер загружаемых данных на несколько сотен килобайтов (или даже больше), улучшая субъективную производительность приложения. Субъективная производительность — то, как пользователь воспринимает производительность приложения; улучшать данный параметр очень важно, особенно если приложение загружается с мобильного устройства, работающего в медленной сети.

3.8. Практикум: добавление навигации в онлайн-аукцион

Упражнение начинается там, где мы остановились в предыдущей главе. К данному моменту вы создали главную страницу аукциона (см. рис. 2.3). Цель проекта заключается в добавлении навигации в приложение, чтобы пользователь мог выбрать название продукта; это вызовет замену представления, показывающего карусель и миниатюры товаров, на представление `ProductItemComponent`.

В настоящей главе вы не увидите финальную версию представления Product Details (Информация о продукте). Несмотря на то, что код, показанный в главе 2, имеет класс `ProductService`, содержащий всю подробную информацию о продукте, мы используем его в главе 4 для иллюстрации внедрения зависимостей. На рис. 3.13 показано, как онлайн-аукцион будет выглядеть в этой главе после того, как пользователь нажмет ссылку `First Product` на главной странице.

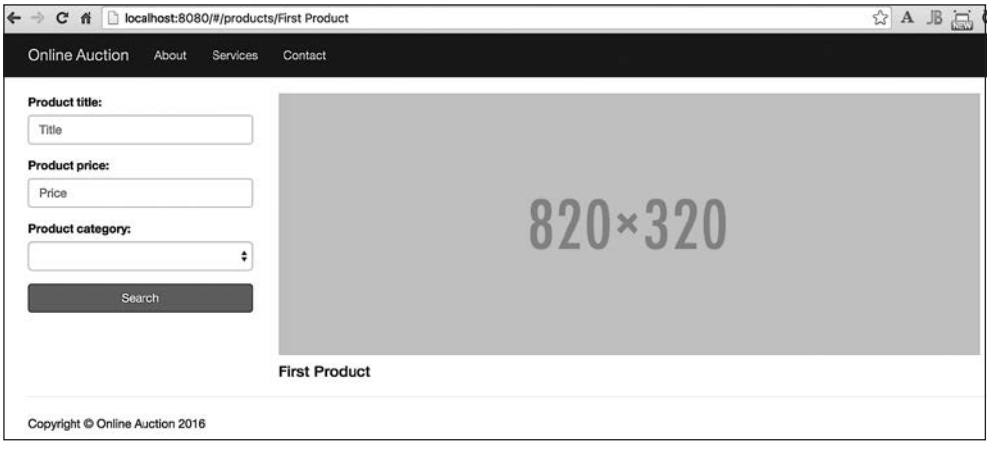


Рис. 3.13. Навигация по маршруту Product Details

В данном разделе нужно сделать следующие шаги.

1. Создать компонент `ProductDetailComponent`, который отображает только заголовок продукта.

2. Переписать код так, чтобы в компоненте `HomeComponent` появились карусель и сетка с элементами, представляющими продукты.
3. Сконфигурировать маршрут для пути `products`, принимающего в качестве аргумента название продукта. Этот маршрут должен переносить пользователя к компоненту `ProductDetailComponent`, который будет принимать в качестве аргумента название продукта с помощью объекта `ActivatedRoute`.
4. Изменить код компонента `ApplicationComponent` так, чтобы тот отрисовывал либо `HomeComponent`, либо `ProductDetailComponent` в зависимости от выбранного маршрута.
5. Добавить тег `<route-outlet>` в основное приложение для отрисовки `HomeComponent` и `ProductDetailComponent`.
6. Добавить ссылку, содержащую `[routerLink]`, в шаблон `ProductItemComponent`, чтобы в момент, когда пользователь нажимает название продукта, приложение переходило по маршруту `Product Details`.

ПРИМЕЧАНИЕ

Если хотите увидеть только финальную версию этого проекта, то откройте каталог auction в командной строке и запустите команды npm install и npm start. Можно также скопировать каталог auction из главы 2 в другое место и следовать инструкциям, приведенным в следующих подразделах.

3.8.1. Создание ProductDetailComponent

Создайте новый каталог `app/components/product-detail` и добавьте в него файл `product-detail.ts` со следующим содержимым (листинг 3.20).

Листинг 3.20. Содержимое файла product-detail.ts

```
import {Component} from '@angular/core';
import {ActivatedRoute} from '@angular/router';

@Component({
  selector: 'auction-product-page',
  template: `
    <div>
      <img src="http://placehold.it/820x320">
      <h4>{{productTitle}}</h4>
    </div>
  `
})
export default class ProductDetailComponent {
  productTitle: string;
  constructor(route: ActivatedRoute){
    this.productTitle = route.snapshot.params['prodTitle'];
  }
}
```

3.8.2. Создание HomeComponent и рефакторинг кода

В главе 2 вы создали главную страницу аукциона, на которой расположились несколько элементов. Нужно переписать код так, чтобы для новой версии главной страницы использовалась маршрутизация. Вы определите область с тегом `<router-outlet>`, где будете отображать либо компонент `HomeComponent`, либо компонент `ProductDetailComponent`. Первый инкапсулирует существующий компонент `CarouselComponent` и сетку, содержащую компонент `ProductItemComponent`. Это делается путем выполнения следующих шагов.

1. Создайте новый каталог `app/components/home` и добавьте в него файл `home.ts` с таким содержимым (листинг 3.21).

Листинг 3.21. Содержимое файла home.ts

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

@Component({
  selector: 'auction-home-page',
  styleUrls: ['/home.css'],
  template: `
    <div class="row carousel-holder">
      <div class="col-md-12">
        <auction-carousel></auction-carousel>
      </div>
    </div>
    <div class="row">
      <div *ngFor="let product of products"
      ➥ class="col-sm-4 col-lg-4 col-md-4">
        <auction-product-item [product]="product">
        ➥ </auction-product-item>
      </div>
    </div>
  `
})
export default class HomeComponent {
  products: Product[] = [];
  constructor(private productService: ProductService) {
    this.products = this.productService.getProducts();
  }
}
```

Angular внедряет в данный компонент `ProductService`, поставщик для этого сервиса объявлен в модуле `AppModule`. О поставщиках вы узнаете из следующей главы.

В главе 2 предыдущий код располагался в файле `application.ts file`. Но вам нужно инкапсулировать этот код внутри компонента `HomeComponent`, а также сконфигурировать маршрут для него в модуле `AppModule`. На следующем шаге вы удалите соответствующий код из файла `application.ts`.

Если видите стили, которые не определены в коде явно, то имейте в виду, что они берутся из CSS, поставляемом вместе с библиотекой Bootstrap. Можете изменять их по мере необходимости.

2. Создайте файл `home.css`, чтобы указать стили для компонента карусели из Bootstrap внутри `HomeComponent` (листинг 3.22).

Листинг 3.22. Содержимое файла home.css

```
.slide-image {
    width: 100%;
}
.carousel-holder {
    margin-bottom: 30px;
}
.carousel-control,.item {
    border-radius: 4px;
}
```

3.8.3. Упрощаем компонент ApplicationComponent

Теперь, когда вы инкапсулировали большой фрагмент кода внутри элемента `HomeComponent`, код компонента `ApplicationComponent` станет короче:

1. Замените содержимое файла `application.ts` на следующий код (листинг 3.23).

Листинг 3.23. Содержимое файла application.ts

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

@Component({
  selector: 'auction-application',
  templateUrl: 'app/components/application/application.html',
  styleUrls: ['app/components/application/application.css'],
  encapsulation:ViewEncapsulation.None
})
export default class ApplicationComponent {}
```

В свойствах `templateUrl` и `styleUrls` вы будете применять полный путь к файлам HTML и CSS. В главе 10 из вставки «Относительные пути в шаблонах при использовании SystemJS» вы узнаете, как задействовать относительные пути при указании файлов HTML и CSS. В рамках книги гораздо проще описать более короткие фрагменты кода, поэтому вы разместите разметку компонента `ApplicationComponent` в отдельном файле `application.html`.

2. Измените содержимое файла `application.html`, чтобы он выглядел следующим образом (листинг 3.24).

Листинг 3.24. Обновленный файл application.html

```
<auction-navbar></auction-navbar>
<div class="container">
  <div class="row">
    <div class="col-md-3">
      <auction-search></auction-search>
    </div>
    <div class="col-md-9">
```

```
      <router-outlet></router-outlet>
    </div>
  </div>
</div>
<auction-footer></auction-footer>
```

Основное изменение заключается в том, что элементы карусели и элементов продуктов заменены на тег `<router-outlet>`. Когда маршрутизатор отрисует компонент `HomeComponent`, будут отрисованы также карусель и элементы продуктов.

В верхней части окна аукционов располагается панель навигации, в нижней — нижний колонтитул, а область посередине разбита надвое: компонент поиска и область отображения маршрутизатора. В соответствии с системой таблицы Bootstrap, вся ширина окна делится на 12 равных колонок, три из них выделяются `<auction-search>`, а еще девять — `<router-outlet>`. Другими словами, 25 % ширины экрана выделяется для поиска, а 75 % — для маршрутов. Вы реализуете функциональность поиска в главе 7, в которой рассматривается работа с формами.

3.8.4. Добавление RouterLink в компонент ProductItemComponent

Компонент `HomeComponent` содержит несколько экземпляров компонента `ProductItemComponent`. Каждый из них должен иметь `routerLink`, чтобы вы могли переходить к компоненту `ProductDetailComponent`, передавая в качестве параметра название продукта. Сделайте следующее.

1. Измените код файла `product-item.ts` для ссылки на файл CSS, как показано здесь (листинг 3.25).

Листинг 3.25. Обновленный файл product-item.ts

```
import {Component, Input} from '@angular/core';
import {Product} from '../../services/product-service';

@Component({
  selector: 'auction-product-item',
  styleUrls: ['app/components/product-item/product-item.css'],
  templateUrl: 'app/components/product-item/product-item.html',
})
export default class ProductItemComponent {
  @Input() product: Product;
}
```

Для файла `product-item.html` требуется якорный тег без директивы `routerLink`, что позволит выполнять навигацию по маршруту, соотнесенному с путем `products/:prodTitle`. Вы сконфигурируете его в модуле `AppModule` несколько позже.

2. Измените содержимое файла `product-item.html`, чтобы он выглядел вот так (листинг 3.26).

Листинг 3.26. Обновленный файл product-item.html

```
<div class="thumbnail">
  <img src="http://placehold.it/320x150">
  <div class="caption">
    <h4 class="pull-right">{{ product.price | currency }}</h4>
    <h4><a [routerLink]="['/
        products', product.title]">{{ product.title }}</a></h4>
    <p>{{ product.description }}</p>
  </div>
  <div class="ratings">
    <auction-stars [rating]="product.rating"></auction-stars>
  </div>
</div>
```

При форматировании цены продукта вы используете канал `currency` (канал указывается после символа вертикальной полосы). Если данный канал не работает в вашем браузере, то в разделе 5.3 вы сможете найти способ обойти эту проблему.

3. Создайте файл `product-item.css`, имеющий следующее содержимое (листинг 3.27).

Листинг 3.27. Содержимое файла product-item.css

```
.caption {
  height: 130px;
  overflow: hidden;
}
.caption h4 { white-space: nowrap;}
.thumbnail { padding: 0;}
.thumbnail img { width: 100%;}
.thumbnail .caption-full {
  padding: 9px;
  color: #333;
}
.ratings {
  color: #d17581;
  padding-left: 10px;
  padding-right: 10px;
}
```

3.8.5. Изменение корневого модуля с целью добавления маршрутизации

Наконец, нужно обновить файл `app.module.ts`, добавив модуль `RouterModule` и стратегию расположения, а также сконфигурировать маршруты (листинг 3.28).

Листинг 3.28. Обновленный файл app.module.ts

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
```

```
import { RouterModule } from '@angular/router';
import {LocationStrategy, HashLocationStrategy} from '@angular/common';
import ApplicationComponent from './components/application/application';
import CarouselComponent from "./components/carousel/carousel";
import FooterComponent from "./components/footer/footer";
import NavbarComponent from "./components/navbar/navbar";
import ProductItemComponent from "./components/product-item/product-item";
import SearchComponent from "./components/search/search";
import StarsComponent from "./components/stars/stars";
import {ProductService} from "./services/product-service";
import HomeComponent from "./components/home/home";
import ProductDetailComponent from "./components/product-detail/
➥ product-detail";

@NgModule({
    imports:      [ BrowserModule,
                    RouterModule.forRoot([
                        {path: '',                    component:
                        ➥ HomeComponent},
                        {path: 'products/:prodTitle',
                        ➥ component: ProductDetailComponent}
    ]) ],
    declarations: [ ApplicationComponent,CarouselComponent,
                    FooterComponent, NavbarComponent,
                    HomeComponent, ProductDetailComponent,
                    ProductItemComponent,SearchComponent,StarsComponent],
    providers:    [ProductService,
                    {provide: LocationStrategy, useClass:
                    ➥ HashLocationStrategy}],
    bootstrap:    [ ApplicationComponent ]
})
export class AppModule { }
```

Здесь вы конфигурируете два маршрута: базовый URL (пустой путь) будет указывать на компонент `HomeComponent`, а путь `products/:prodTitle` нужен для отрисовки компонента `ProductDetailComponent`, получающего в качестве параметра название продукта. Значение `prodTitle` будет предоставлено компонентом `ProductItemComponent`, в котором был определен `routerLink`.

3.8.6. Запуск аукциона

Переключитесь в каталог `auction` в командной строке и запустите сервер, введя команду `npm start` (сценарий запуска сконфигурирован в файле `package.json`, как показано в главе 2). Браузер откроет главную страницу аукциона, которая будет выглядеть аналогично изображению в главе 2. Теперь нажмите название любого из продуктов, вы должны увидеть упрощенную страницу `Product Details`, показанную на рис. 3.13. В главе 4 вы внедрите дополнительную информацию о продуктах в это представление.

Применение оператора расширения

По мере роста приложения количество компонентов, объявленных вами в модуле `AppModule`, может сделать код менее читаемым. Использование оператора расширения ES6 (рассматривается в приложении А) может помочь в этой ситуации. Создайте отдельный файл, в котором вы перечислите все свои компоненты, например, так:

```
export const myComponents = [
    ApplicationComponent,
    CarouselComponent,
    FooterComponent,
    NavbarComponent,
    HomeComponent,
    ProductDetailComponent,
    ProductItemComponent,
    SearchComponent,
    StarsComponent];
```

Далее декоратор `@NgModule` сможет применить оператор расширения следующим образом:

```
@NgModule({
    // здесь будет другой код
    declarations: [ ...myComponents],
    // здесь будет другой код
})
```

3.9. Резюме

Из этой главы вы узнали, как реализовать навигацию в одностраничном приложении с помощью маршрутизатора Angular. Вот основные выводы.

- Конфигурируйте маршруты для вашего приложения с помощью `RouterModule`.
- Выбирая стратегию расположения, вы можете управлять тем, как выглядит URL для каждого представления.
- При навигации по приложению маршрутизатор отрисовывает лежащий в основе маршрута компонент в области содержимого, определенной тегами `<router-outlet>`. У вас может быть больше одной такой области.
- Чтобы выполнить навигацию по маршруту, добавьте в приложение якорные теги. Они должны использовать свойство `routerLink` вместо атрибута `href`. После этого вы сможете передавать параметры в маршрут.
- Для уменьшения размера приложения проверьте, можно ли некоторые модули загружать отдельно по требованию, реализовав ленивую загрузку.

4 Внедрение зависимостей

В этой главе:

- знакомство с шаблоном проектирования «Внедрение зависимостей» (Dependency Injection, DI);
- преимущества DI;
- реализация DI в Angular;
- регистрация поставщиков объектов и использование инъекторов;
- иерархия инъекторов;
- применение DI в приложении онлайн-аукциона.

В предыдущей главе мы рассмотрели маршрутизатор, и теперь ваше приложение-аукцион знает, как перейти от представления Home (Главная страница) к почти пустому представлению Product Details (Информация о продукте). В данной главе вы продолжите работу над онлайн-аукционом, но в этот раз сконцентрируете усилия на автоматизации процесса создания объектов и сборки приложения из составных частей.

Любое Angular-приложение представляет собой коллекцию объектов, директив и классов, которые могут зависеть друг от друга. Несмотря на то, что каждый компонент может явно создавать экземпляры своих зависимостей, Angular способен выполнять эту задачу с помощью механизма внедрения зависимостей.

Мы начнем эту главу с того, что определим, какие проблемы решает DI, и рассмотрим преимущества DI как шаблона проектирования. Далее рассмотрим, как данный шаблон реализуется в Angular на примере компонента `ProductComponent`, который зависит от `ProductService`. Вы увидите, как написать внедряемый сервис и внедрить его в другой элемент.

Далее мы представим пример приложения, демонстрирующий, как Angular DI позволяет с легкостью заменять одну зависимость другой, изменяя всего одну строку кода. После этого познакомимся с более продвинутой концепцией: иерархией инъекторов.

В конце главы мы создадим новую версию онлайн-аукциона, в которой используются приемы, описанные в данной главе.

4.1. Шаблоны «Внедрение зависимостей» и «Инверсия управления»

Шаблоны проектирования — рекомендации по решению некоторых распространенных задач. Заданный шаблон проектирования может быть реализован разными способами в зависимости от используемого ПО. В этом разделе мы кратко рассмотрим два шаблона проектирования: «Внедрение зависимостей» (Dependency Injection, DI) и «Инверсия управления» (Inversion of Control, IoC).

4.1.1. Шаблон «Внедрение зависимостей»

Если вы когда-нибудь писали функцию, принимающую в качестве аргумента объект, то можете сказать, что писали программу, которая создает объект и внедряет его в функцию. Представьте фулфилмент-центр, отправляющий продукты. Приложение, которое отслеживает отправленные продукты, может создать объект продукта и вызвать функцию, создающую и сохраняющую запись об отправке:

```
var product = new Product();
createShipment(product);
```

Функция `createShipment()` зависит от существования экземпляра класса `Product`. Другими словами, функция `createShipment()` имеет зависимость: `Product`. Но сама по себе она не знает, как создавать объекты такого типа. Вызывающий сценарий должен каким-то образом создавать и передавать (то есть внедрять) этот объект как аргумент функции. Технически, вы отвязываете место создания объекта `Product` от места его использования — но обе предыдущие строки кода находятся в одном сценарии, поэтому данное отвязывание нельзя назвать настоящим. При необходимости заменить тип `Product` на тип `MockProduct` понадобится внести небольшое изменение в наш простой пример.

Что если функция `createShipment()` имеет три зависимости (продукт, компанию-отправителя, фулфилмент-центр), и каждая из них имеет собственные зависимости? В таком случае для создания разного набора объектов для функции `createShipment()` потребуется внести большое количество изменений. Можно ли попросить кого-то создать экземпляры зависимостей (и их зависимостей) вместо вас?

Здесь и нужен шаблон «Внедрение зависимостей»: если объект А зависит от объекта типа Б, то объект А не будет явно создавать объект Б (в случае использования оператора `new`, как в предыдущем примере). Вместо этого объект Б будет внедрен из операционной среды. Объект А просто должен объявить следующее: «Мне нужен объект типа Б; может ли кто-то его мне передать?» Слово «типа» здесь самое важное. Объект А не запрашивает конкретную реализацию объекта, и его запрос будет удовлетворен, если внедряемый объект имеет тип Б.

4.1.2. Шаблон «Инверсия управления»

Шаблон «Инверсия управления» является более общим, нежели DI. Вместо того чтобы использовать в своем приложении API фреймворка (или программного контейнера), фреймворк создает и отправляет объекты, необходимые приложению. Шаблон IoC может быть реализован разными способами, а DI — это один из способов предоставления требуемых объектов. Angular играет роль контейнера IoC и может предоставлять требуемые объекты в соответствии с объявлениями, сделанными в вашем компоненте.

4.1.3. Преимущества внедрения зависимости

Прежде чем исследовать синтаксис реализации шаблона DI в Angular, рассмотрим преимущества внедрения объектов перед созданием их с помощью оператора `new`. Angular предлагает механизм, помогающий регистрировать и создавать экземпляры компонентов, для которых имеется зависимость. Короче говоря, DI позволяет писать слабо связанный код, а также применять его повторно и проводить более качественное тестирование.

Слабое связывание и повторное использование

Предположим, у вас имеется компонент `ProductComponent`, который получает подробную информацию о продукте с помощью класса `ProductService`.

Если вы не используете DI, то компонент `ProductComponent` должен знать, как создавать объекты класса `ProductService`. Это можно сделать несколькими способами, например, задействовать оператор `new`, вызвать метод `getInstance()` для объекта-синглтона или вызвать метод `createProductService()` какого-нибудь класса-фабрики. В любом из описанных случаев компонент `ProductComponent` становится *тесно (жестко) связанным* с классом `ProductService`.

Если вам нужно использовать компонент `ProductComponent` в другом приложении, которое применяет другой сервис для получения подробной информации о продукте, то вы должны модифицировать код (например, так: `productService = new AnotherProductService()`). DI позволяет отвязать компоненты приложения, избавив их от необходимости знать, как создавать зависимость.

Рассмотрим следующий пример компонента `ProductComponent`:

```
@Component({
  providers: [ProductService]
})
class ProductComponent {
  product: Product;
  constructor(productService: ProductService) {
    this.product = productService.getProduct();
  }
}
```

В приложениях вы регистрируете объекты для DI, указывая поставщики. *Поставщик* — это инструкция для Angular о том, *как* создать экземпляр объекта для последующего внедрения в целевой компонент или директиву. В предыдущем фрагменте кода строка `providers:[ProductService]` является сокращением строки `providers:[{provide:ProductService, useClass:ProductService}]`.

ПРИМЕЧАНИЕ

Вы видели свойство providers в главе 3, но оно было реализовано на уровне модуля, а не компонента.

Angular применяет концепцию токенов — произвольных имен, представляющих внедряемый объект. Обычно имя токена соответствует типу внедряемого объекта, соответственно, в предыдущем фрагменте кода Angular получает указание предоставить токен `ProductService`, задействуя класс с тем же именем. Использование объекта со свойством `provide` позволяет соотнести один токен с разными значениями или объектами (например, эмулировать функциональность класса `ProductService`, когда кто-то разрабатывает реальный класс сервиса).

ПРИМЕЧАНИЕ

В подразделе 4.4.1 вы увидите, как можно объявить токен с произвольным именем.

Теперь, когда вы добавили свойство `providers` в аннотацию `@Component` компонента `ProductComponent`, модуль DI, предоставляемый Angular, будет знать, что он должен создать объект *типа* `ProductService`. Компонент `ProductComponent` не должен знать, какую именно реализацию типа `ProductService` будет использовать — он применит любой объект, указанный как поставщик. Ссылка на объект типа `ProductService` будет внедрена с помощью аргумента конструктора, нет необходимости явно создавать объект типа `ProductService` в компоненте `ProductComponent`. Просто задействуйте его как предыдущий код, который вызывает метод сервиса `getProduct()` экземпляра класса `ProductService`, созданного Angular.

Если нужно использовать один и тот же компонент `ProductComponent` в разных приложениях, имеющих разную реализацию типа `ProductService`, то измените строку providers, как показано в следующем примере:

```
providers: [{provide: ProductService, useClass: AnotherProductService}]
```

Теперь Angular будет создавать экземпляр класса `AnotherProductService`, но код, задействующий тип `ProductService`, не будет генерировать ошибки. В этом примере использование DI увеличивает возможность повторного применения компонента `ProductComponent` и разрушает тесное связывание с классом `ProductService`. Если один объект тесно связан с другим, то для использования хотя бы одного из них может потребоваться внести много изменений.

Тестируемость

DI позволяет более качественно тестировать компоненты отдельно друг от друга. Можно легко внедрять фальшивые объекты, если их реальные реализации недоступны или когда нужно организовать модульное тестирование кода.

Предположим, вам нужно добавить в приложение возможность авторизации. Можно создать компонент `LoginComponent` (для отрисовки полей `ID` и `password`), использующий компонент `LoginService`, который должен соединяться с определенным сервером авторизации и проверять привилегии пользователя. Сервер авторизации должен быть предоставлен другим отделом, но он еще не готов. Вы завершаете написание кода компонента `LoginComponent`, но затрудняетесь протестировать его по причинам, которые не можете контролировать, например, из-за зависимости или другого компонента, разрабатываемого другими людьми.

При тестировании часто применяются фальшивые объекты, имитирующие поведение реальных. В случае использования фреймворка DI можно создать фальшивый объект, `MockLoginService`, который не соединяется с сервером авторизации, но при этом в нем жестко закодированы привилегии, присвоенные пользователям, имеющим определенные комбинации идентификатора и пароля. Задействуя DI, можно написать всего одну строку, в которой `MockLoginService` будет внедрен в представление Login (Авторизация) приложения, что позволит не ждать готовности сервера авторизации. Далее, когда сервер будет готов, можно изменить строку `providers` так, чтобы Angular внедрил реальный компонент `LoginService`, как показано на рис. 4.1.

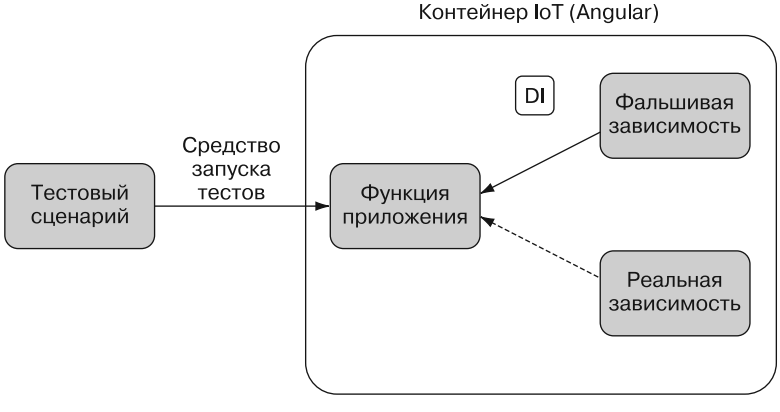


Рис. 4.1. DI при тестировании

ПРИМЕЧАНИЕ

В разделе «Практикум» главы 9 вы увидите, как выполнять модульное тестирование внедряемых сервисов.

4.2. Инъекторы и поставщики

Теперь, когда вы кратко ознакомились с шаблоном «Внедрение зависимости», перейдем к деталям реализации DI в Angular. В частности, мы рассмотрим такие концепции, как инъекторы и поставщики.

Каждый компонент может иметь объект `Injector`, который может внедрять объекты и примитивные значения в элемент или сервис. В любом приложении Angular имеется корневой инъектор, доступный всем его модулям. Чтобы указать инъектору, *что именно* внедрить, вы указываете поставщик. Инъектор внедрит объект или значение, указанное в поставщике, в конструктор компонента.

ПРИМЕЧАНИЕ

Несмотря на то, что мгновенно загруженные модули не имеют собственных инъекторов, модули, загруженные лениво, имеют собственный подкорневой инъектор, который является непосредственным потомком корневого инъектора приложения.

Поставщики позволяют соотносить пользовательские типы (или токены) с конкретными реализациями этого типа (или значениями). Можно указать поставщики либо внутри декоратора компонента `@Component`, либо как свойство `@NgModule`, что было сделано в каждом фрагменте кода, представленного до настоящего момента.

СОВЕТ

В Angular вы можете внедрять данные только с помощью аргументов конструктора. Если вы видите класс, чей конструктор не имеет аргументов, то гарантированно не сможете ничего внедрить в этот компонент.

Вы будете использовать компонент `ProductComponent` и класс `ProductService` во всех примерах кода, показанных в этой главе. Если ваше приложение имеет класс, реализующий определенный тип (например, `ProductService`), то вы можете указать объект поставщика для данного класса во время предварительной загрузки модуля `AppModule`, например, так:

```
@NgModule({
  ...
  providers: [{provide:ProductService,useClass:ProductService}]
})
```

Если имя токена совпадает с именем класса, то можно использовать более короткую нотацию, чтобы указать поставщик в модуле:

```
@NgModule({
  ...
  providers: [ProductService]
})
```

Можно указать свойство `providers` в аннотации `@Component`. Короткая нотация поставщика `ProductService` в `@Component` выглядит так:

```
providers:[ProductService]
```

Ни один экземпляр типа `ProductService` еще не был создан. Строка `providers` указывает инъектору следующее: «Когда нужно создать объект, имеющий аргумент типа `ProductService`, создайте экземпляр зарегистрированного класса для внедрения в этот объект».

ПРИМЕЧАНИЕ

Angular также имеет свойство `viewProviders`, которое задействуется, когда вы не хотите, чтобы компоненты-потомки применяли поставщики, объявленные предками. Вы увидите пример использования `viewProviders` в разделе 4.5.

Если нужно внедрить разные реализации определенного типа, примените более длинную нотацию:

```
@NgModule({
  ...
  providers: [{provide:ProductService,useClass:MockProductService}]
})
```

Так она выглядит на уровне компонента:

```
@Component({
  ...
  providers: [{provide:ProductService, useClass:MockProductService}]
})
```

Она дает инъектору следующую инструкцию: «Когда нужно внедрить объект типа `ProductService` в компонент, создайте экземпляр класса `MockProductService`».

Благодаря поставщику инъектор знает, *что* внедрять; теперь нужно указать, *куда* внедрять объект. В TypeScript все сводится к объявлению аргумента конструктора с указанием его типа. Следующая строка показывает, как внедрить объект типа `ProductService` в конструктор компонента:

```
constructor(productService: ProductService)
```

Внедрение: TypeScript против ES6

В TypeScript упрощен синтаксис внедрения в компонент, поскольку этот язык не требует использования аннотаций DI для аргументов конструкторов. Все, что нужно сделать — указать тип аргумента конструктора:

```
constructor(productService: ProductService)
```

Этот код работает, потому что любой компонент имеет аннотацию `@Component`. И, поскольку компилятор TypeScript сконфигурирован с настройкой `"emitDecoratorMetadata": true`, Angular автоматически сгенерирует все метаданные, необходимые для того, чтобы внедрить объект.

Так как вы используете SystemJS для динамической компиляции в код TypeScript, можете добавить следующую настройку компилятора TypeScript в файле `systemjs.config.js`:

```
typescriptOptions: {
    "emitDecoratorMetadata": true
}
```

Если вы пишете этот класс в ES6, то добавьте аннотацию `@Inject` и явный тип в аргументы конструктора:

```
constructor(@Inject(ProductService) productService)
```

Конструктор останется таким же независимо от того, какая конкретная реализация класса `ProductService` будет указана в качестве поставщика. На рис. 4.2 показана примерная схема последовательности процесса внедрения.

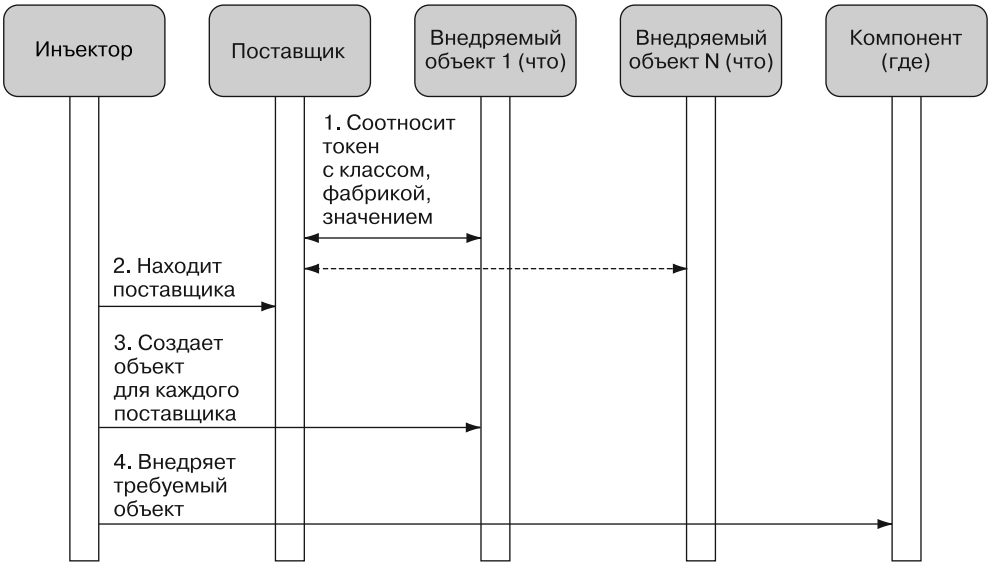


Рис. 4.2. Процесс внедрения

4.2.1. Как объявлять поставщики

Можно объявить пользовательские поставщики как массив объектов, содержащий свойство `provide`. Такой массив может быть указан в свойстве `providers` модуля или на уровне компонента.

Рассмотрим пример массива с одним элементом, в котором указан объект поставщика для токена `ProductService`:

```
[{provide:ProductService, useClass:MockProductService}]
```

Свойство `provide` позволяет соотнести токен с методом, создающим внедряемый объект. В этом примере вы даете Angular указание создать объект класса `MockProductService` там, где в качестве зависимости используется токен `ProductService`. Но создатель объекта (инъектор Angular) может применять класс, функцию фабрики, строку или специальный класс `OpaqueToken` для создания объекта или его внедрения.

- Чтобы соотнести токен и реализацию класса, используйте объект, имеющий свойство `useClass`, как показано в предыдущем примере.
- При наличии функции фабрики, создающей объекты на основе определенных критериев, задействуйте объект со свойством `useFactory`. Оно позволяет указать функцию фабрики (или анонимное стрелочное выражение), которая знает, как создавать требуемые объекты. Такая функция может иметь необязательный аргумент с зависимостями, если они существуют.
- Чтобы предоставить строку с простым внедряемым значением (например, URL сервиса), обратитесь к объекту со свойством `useValue`.

В следующем разделе вы будете применять свойство `useClass`, а также изучите простое приложение. В разделе 4.4 будет показано использование свойств `useFactory` и `useValue`.

4.3. Пример приложения, задействующего Angular DI

Теперь, когда вы увидели несколько фрагментов кода, связанных с Angular DI, создадим небольшое приложение, которое объединяет все фрагменты воедино. Мы хотим подготовить вас к использованию DI в приложении для онлайн-аукциона.

4.3.1. Внедрение сервиса продукта

Создадим простое приложение, применяющее компонент `ProductComponent` для отрисовки информации о продукте и сервис `ProductService`, который предоставляет данные о продукте. Если вы используете загружаемый код, поставляемый с книгой, то данное приложение находится в файле `main-basic.ts` в каталоге `di_samples`. В этом подразделе вы создадите приложение, генерирующее страницу, показанную на рис. 4.3.

Basic Dependency Injection Sample

Product Details

Title: iPhone 7

Description: The latest iPhone, 7-inch screen

Price: $249.99

Рис. 4.3. Пример приложения, использующего DI

Компонент `ProductComponent` может запросить внедрение объекта `ProductService` путем объявления аргумента конструктора с типом:

```
constructor(productService: ProductService)
```

На рис. 4.4 показан пример приложения, которое использует эти компоненты.

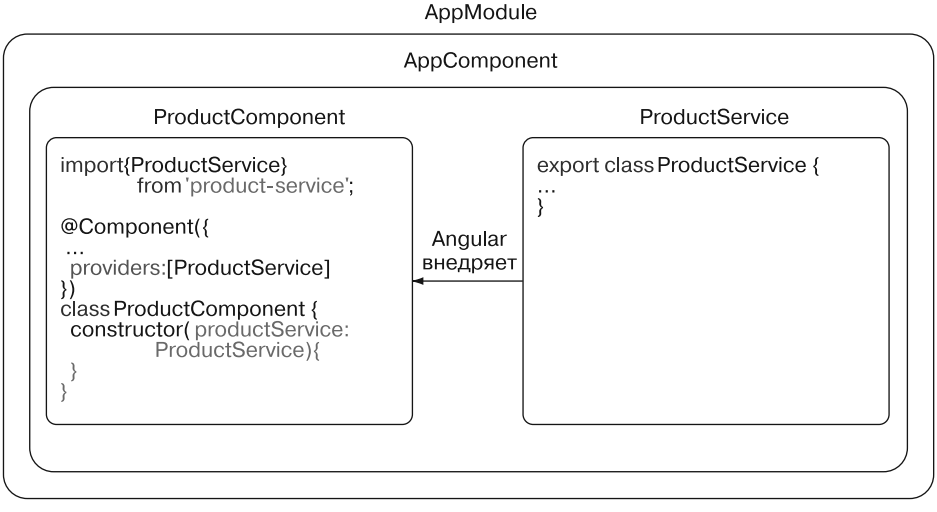


Рис. 4.4. Внедрение ProductService в ProductComponent

Модуль `AppModule` предварительно загружает `AppComponent`, содержащий компонент, зависящий от `ProductService`. Обратите внимание на операторы импорта и экспорта. Определение `ProductService` начинается с оператора экспорта, что позволяет другим элементам получить доступ к его содержимому. Компонент `ProductComponent` содержит оператор импорта, который предоставляет имя класса (`ProductService`) и импортируемого модуля (располагается в файле `product-service.ts`).

Атрибут `providers`, определенный на уровне компонента, указывает Angular предоставить экземпляр класса `ProductService` по требованию. Данный класс может общаться с каким-нибудь сервером, запрашивая подробную информацию о продукте, выбранном на веб-странице. Но мы сейчас опустим эту часть и сконцентрируемся на том, как указанный сервис можно внедрить в `ProductComponent`. Реализуем компоненты, показанные на рис. 4.4.

Помимо `index.html` вы создадите следующие файлы:

- файл `main-basic.ts` будет содержать код, необходимый для загрузки модуля `AppModule`, который содержит компонент `AppComponent`, размещающий `ProductComponent`;
- компонент `ProductComponent` будет реализован в файле `product.ts`;
- сервис `ProductService` будет реализован в файле `product-service.ts`.

Каждый из этих файлов довольно прост. Файл `main-basic.ts`, показанный в листинге 4.1, содержит код модуля и корневой компонент, который размещает компонент-потомок `ProductComponent`. Этот модуль импортирует и объявляет данный компонент.

Листинг 4.1. Содержимое файла main-basic.ts

```
import {Component} from '@angular/core';
import ProductComponent from './components/product';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

@Component({
    selector: 'app',
    template: `<h1> Basic Dependency Injection Sample</h1>
            <di-product-page></di-product-page>`
})
class AppComponent {}

@NgModule({
    imports:      [ BrowserModule],
    declarations: [ AppComponent, ProductComponent],
    bootstrap:    [ AppComponent ]
})
class AppModule { }
platformBrowserDynamic().bootstrapModule(AppModule);
```

Основываясь на теге `<di-product-page>`, легко догадаться, что существует элемент с селектором, имеющим это значение. Данный селектор объявлен в компоненте `ProductComponent`, чья зависимость (`ProductService`) внедрена с помощью конструктора (листинг 4.2).

Листинг 4.2. Содержимое файла product.ts

```
import {Component, bind} from '@angular/core';
import {ProductService, Product} from "../services/product-service";

@Component({
  selector: 'di-product-page',
  template: `<div>
  <h1>Product Details</h1>
  <h2>Title: {{product.title}}</h2>
  <h2>Description: {{product.description}}</h2>
  <h2>Price: \${{product.price}}</h2>
</div>`,
  providers:[ProductService]
})

export default class ProductComponent {
  product: Product;

  constructor( productService: ProductService) {

    this.product = productService.getProduct();
  }
}
```

Короткая нотация свойства providers указывает инъектору создать объект класса ProductService

Angular создает объект класса ProductService и внедряет его сюда

В листинге 4.2 имя типа совпадает с именем класса — `ProductService`, так что можно использовать короткую нотацию без необходимости явно соотносить свойства

provide и useClass. При указании поставщиков имя (токен) внедряемого объекта отделяется от его реализации. В этом случае имя токена будет таким же, как и имя типа: ProductService. Сама реализация данного сервиса может находиться в классах ProductService, OtherProductService или где-то еще. Замена одной реализации на другую сводится к изменению строки providers.

Конструктор компонента ProductComponent вызывает метод getProduct() для сервиса и размещает ссылку на возвращенный объект типа Product в переменной класса продукта, которая применяется в шаблоне HTML. Используя двойные фигурные скобки, в листинге 4.2 можно связать свойства title, description и price класса Product.

Файл product-service.ts содержит определение двух классов: Product и ProductService (листинг 4.3).

Листинг 4.3. Содержимое файла product-service.ts

```
export class Product {
  constructor(
    public id: number,
    public title: string,
    public price: number,
    public description: string) {
  }
}
export class ProductService {

  getProduct(): Product {
    return new Product(0, "iPhone 7", 249.99, "The latest iPhone,
    ➥ 7-inch screen");
  }
}
```

Класс Product представляет собой продукт (объект value). Он используется за пределами данного сценария, поэтому вы экспортируете его

Для простоты метод getProduct() будет возвращать один и тот же продукт, чьи значения были жестко закодированы

В реальных приложениях метод getProduct() должен будет получать информацию о продукте из внешнего источника данных, например, отправляя HTTP-запрос на удаленный сервер.

Чтобы запустить этот пример, откройте командную строку в каталоге проекта и выполните команду npm start. Live-сервер откроет окно, как было показано ранее на рис. 4.3. Экземпляр класса ProductService внедрен в компонент ProductComponent, который отрисовывает информацию о продукте, предоставленную сервером.

В следующем разделе вы увидите сервис ProductService, декорированный аннотацией @Injectable. Его можно использовать для генерации метаданных для DI, если сервис сам по себе имеет зависимости. Аннотация @Injectable здесь не нужна, поскольку в данный сервис не внедрены другие сервисы, и Angular не нужны дополнительные метаданные для его внедрения в свои компоненты.

4.3.2. Внедрение Http-сервиса

Зачастую сервису нужно сделать HTTP-запрос, чтобы получить необходимые данные. Компонент ProductComponent зависит от сервиса ProductService, внедренного с помощью механизма Angular DI. Если данному сервису нужно сделать HTTP-запрос, то он будет иметь в качестве зависимости объект Http. Для внедрения

объекта `Http` сервису `ProductService` нужно будет его импортировать; аннотация `@NgModule` должна импортировать модуль `HttpModule`, в котором определены поставщики `Http`. Класс `ProductService` должен иметь конструктор для внедрения объекта `Http`. На рис. 4.5 показано, что компонент `ProductComponent` зависит от сервиса `ProductService`, который имеет собственную зависимость — `Http`.

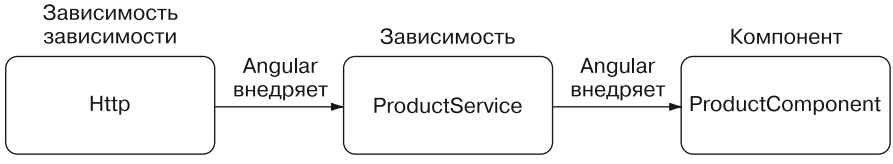


Рис. 4.5. Зависимость может иметь собственную зависимость

В следующем фрагменте кода показано внедрение объекта `Http` в сервис `ProductService`, а также получение продуктов из файла `products.json`:

```
import {Http} from '@angular/http';
import {Injectable} from "@angular/core";

@Injectable()
export class ProductService {
    constructor(private http:Http){
      let products = http.get('products.json');
    }
    // здесь будет располагаться другой код приложения
}
```

Точкой внедрения будет являться конструктор класса, но где же объявить поставщика для внедрения объекта типа `Http`? Все поставщики должны внедрять разнообразные объекты `Http`, объявленные в модуле `HttpModule`. Нужно лишь добавить его в модуль `AppModule`, например, так:

```
import { HttpModule} from '@angular/http';
...
@NgModule({
  imports: [
    BrowserModule,
    HttpModule
  ],
  declarations: [ AppComponent ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

ПРИМЕЧАНИЕ

В подразделе 8.3.4 вы напишете приложение, в котором будет проиллюстрирована архитектура, показанная на рис. 4.5.

Теперь, когда вы увидели способы внедрения объекта в компонент, взглянем на то, как заменить одну реализацию сервиса другой с помощью Angular DI.

4.4. Переключение внедряемых объектов — это просто

Ранее в данной главе мы указали, что шаблон DI позволяет отвязать компоненты от их зависимостей. В предыдущем разделе вы отвязали компонент `ProductComponent` от сервиса `ProductService`. Теперь симулируем другой сценарий.

Предположим, вы начали разработку, имея сервис `ProductService`, который должен получать данные от сервера, но сам сервер еще не готов. Вместо того чтобы изменять код сервиса `ProductService`, добавляя туда жестко закодированные значения, вы создадите другой класс: `MockProductService`.

Для демонстрации того, как просто можно переключиться с одного сервиса на другой, вы также создадите небольшое приложение, которое задействует два экземпляра компонента `ProductComponent`. Изначально первый из них будет применять сервис `MockProductService`, а второй — сервис `ProductService`.

Далее, изменив всего одну строку, вы дадите им указание использовать один и тот же сервис.

На рис. 4.6 показано, как несколько приложений _injectors отрисовывают компоненты продукта в браузере.

Рис. 4.6. Отрисовка двух продуктов

Продукт iPhone 7 отрисовывается компонентом `Product1Component`, а продукт Samsung 7 — `Product2Component`. Данное приложение концентрируется на переключении сервисов продуктов с помощью Angular DI, поэтому сами сервисы мы сделали простыми. По той же причине весь код TypeScript находится в одном файле — `main.ts`.

Класс, играющий роль интерфейса

В приложении Б мы объясним интерфейсы TypeScript, которые представляют собой способ гарантировать: объект, переданный функции, корректен; или класс, реализующий интерфейс, следует объявленному контракту. Класс может реализовывать интерфейс с помощью ключевого слова `implements`, но в TypeScript все классы могут быть использованы как интерфейсы (однако мы не поощряем применение данной функциональности), поэтому `ClassA` может реализовывать `ClassB`. Даже если код изначально был написан без интерфейсов, вы все еще можете задействовать конкретный класс в качестве интерфейса.

Содержимое файла `main.ts` показано в листинге 4.4. Мы хотели бы обратить ваше внимание на следующую строку:

```
class MockProductService implements ProductService
```

В ней показан один класс, «реализующий» другой, как если бы последний был объявлен в качестве интерфейса.

Листинг 4.4. Содержимое файла main.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule, Component } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser'

class Product {
  constructor(public title: string) {}
}

class ProductService {
  getProduct(): Product {
    // Здесь будет размещаться код, делающий HTTP-запрос
    // для получения подробной информации о продукте
    return new Product('iPhone 7');
  }
}

class MockProductService implements ProductService {
  getProduct(): Product {
    return new Product('Samsung 7');
  }
}

@Component({
  selector: 'product1',
  template: '{{product.title}}'})
class Product1Component {
  product: Product;

  constructor(private productService: ProductService) {
    this.product = productService.getProduct();
  }
}

@Component({
  selector: 'product2',
  template: '{{product.title}}',
  providers: [{provide:ProductService, useClass:MockProductService}]
})
class Product2Component {
  product: Product;

  constructor(private productService: ProductService) {
    this.product = productService.getProduct();
  }
}
```

Изначально вы разработали ProductService как класс

Далее добавили еще один сервис, MockProductService, который реализует ProductService как интерфейс

В конструктор компонента ProductComponent1 внедряется экземпляр сервиса ProductService

Объявляет конкретную реализацию сервиса ProductService во втором компоненте

Изменять конструктор не нужно. Компонент ProductComponent2 получает MockProductService, поскольку его поставщик был указан на уровне компонента

```
@Component({
  selector: 'app',
  template: `
    <h2>A root component hosts two products<br>
    ➥ provided by different services</h2>
    <product1></product1>
    <br>
    <product2></product2>
  `
})
class AppComponent {}

@NgModule({
  imports: [ BrowserModule],
  providers: [ProductService],
  declarations: [ AppComponent, Product1Component, Product2Component],
  bootstrap: [ AppComponent ]
})
class AppModule { }

platformBrowserDynamic().bootstrapModule(AppModule);
```

Компонент AppComponent отрисовывает два компонента-потомка, каждый из которых использует разные экземпляры сервиса ProductService

Регистрирует поставщика, имеющего инъектор на уровне приложения

Если для компонента не нужна конкретная реализация `ProductService`, то не нужно явно объявлять поставщик для каждого элемента до тех пор, пока поставщик объявлен на уровне предка. В листинге 4.4 для компонента `Product1Component` не объявлены собственные поставщики, поэтому Angular найдет поставщик на уровне приложения. Но каждый элемент может переопределять объявления поставщиков, сделанные на уровне приложения или родительского компонента, как в случае с `Product2Component`.

Сервис `ProductService` становится общим токеном, который понимают оба компонента. В `Product2Component` явно объявляется поставщик, соотносящий `MockProductService` с общим пользовательским типом `ProductService`. Этот поставщик уровня компонента переопределит родительский поставщик. Если вы решите, что `Product1Component` также должен использовать `MockProductService`, то можете добавить строку `providers` в аннотацию `@Component`, как сделано в `Product2Component`.

При запуске данного приложения в браузере будут отрисованы компоненты продукта, как было показано ранее на рис. 4.6. Это все хорошо, но предположим: другая команда сказала вам, что класс `ProductService` (использованный в качестве поставщика на уровне приложения) не будет доступен некоторое время. Как переключиться на применение исключительно класса `MockProductService` в данный временной промежуток?

Для этого нужно изменить одну строку — `providers` в объявлении модуля:

```
@NgModule({
...
providers: [{provide:ProductService, useClass:MockProductService}]
...
})
```

С этого момента, если понадобится внедрить тип `ProductService` и на уровне компонента не будет указано никаких поставщиков, Angular создаст и внедрит объект `MockProductService`. Запуск приложения после внесения вышеозначенных правок изменит отрисовку элементов так, как показано на рис. 4.7.

Рис. 4.7. Отрисовка двух продуктов с помощью `MockProductService`

Представьте, что ваше приложение имеет десяток компонентов, использующих сервис `ProductService`. Если каждый из них будет создавать этот сервис с помощью оператора `new` или класса-фабрики, то вам придется внести десятки изменений в код. Механизм Angular DI поможет исправить применяемый сервис, изменив одну строку в объявлении поставщиков.

Поднятие и классы JavaScript

Объявления классов не подняты (этот процесс объясняется в приложении А). Как правило, каждый класс создается в отдельном файле, и их объявления импортируются в начале работы сценария, что делает их доступными заранее.

При объявлении нескольких классов в одном файле сервисы `ProductService` и `MockProductService` должны быть объявлены до того, как будут объявлены компоненты, которые их используют. Если вы столкнетесь с ситуацией, когда объекты объявляются после точки внедрения, то рассмотрите возможность использовать функцию `forwardRef()` с аннотацией `@Inject` (см. документацию для Angular по функции `forwardRef()` на https://angular.io/api/core/forwardRef).

4.4.1. Объявление поставщиков с помощью useFactory и useValue

Рассмотрим несколько примеров, иллюстрирующих применение поставщиков `factory` и `value`. Как правило, функции-фабрики используются, когда надо реализовать логику приложения до создания объекта. Например, нужно решить заранее, какой объект создавать, или объект может иметь конструктор с аргументами, которые следует инициализировать прежде, чем этот объект будет создан.

В листинге 4.5, взятом из файла `main-factory.ts`, показано, как можно указать в качестве поставщика функцию-фабрику. Она создает объекты сервиса — `ProductService` либо `MockProductService`, — основываясь на булевом флаге.

Листинг 4.5. Указание функции-фабрики в качестве поставщика

```
const IS_DEV_ENVIRONMENT: boolean = true;

@Component({
  selector: 'product2',
  providers:[{
    provide: ProductService,
    useFactory: (isDev) => {
      if (isDev){
        return new MockProductService();
      } else{
        return new ProductService();
      }
    },
    deps: ["IS_DEV_ENVIRONMENT"]}],
  template: '{{product.title}}'
})
class Product2Component {
  product: Product;
  constructor(productService: ProductService) {
    this.product = productService.getProduct();
  }
}
```

Сначала вы определяете токен с произвольным именем (в нашем случае `IS_DEV_ENVIRONMENT`) и устанавливаете его значение равным `true` для извещения программы о том, что работаете в среде разработки (то есть хотите работать с фальшивым сервисом создания продуктов). Фабрика использует анонимное выражение, которое создаст объект класса `MockProductService`.

Конструктор компонента `Product2Component` имеет аргумент типа `ProductService`, куда будет внедрен сервис. Вы можете использовать такую фабрику и для `Product1Component`; изменение значения `IS_DEV_ENVIRONMENT` на `false` внедрит сервис `ProductService` в оба элемента.

В листинге 4.5 показано не лучшее решение по переключению сред: в нем вы обращаетесь к флагу `IS_DEV_ENVIRONMENT`, который объявлен за пределами компонента; это нарушает его инкапсуляцию. Вы хотите, чтобы компонент был самостоятельным, так что попробуем внедрить в него значение переменной `IS_DEV_ENVIRONMENT`; в данном случае вам не понадобится обращаться ко внешнему коду.

Чтобы внедрить значение, недостаточно просто объявить константу (или переменную). Нужно зарегистрировать значение `IS_DEV_ENVIRONMENT` в инъекторе, используя `provide` с `useValue`; это позволит использовать его как внедряемый параметр в анонимное выражение из листинга 4.5.

ПРИМЕЧАНИЕ

Свойства useFactory и useValue находятся в Angular Core. Свойство useValue представляет собой особый случай useFactory, когда фабрика представлена одним выражением и не нуждается в других зависимостях.

Для легкого переключения между средой разработки и другими средами можно указать поставщик значения для среды на корневом уровне компонента, как показано в листинге 4.6; благодаря этому фабрика сервисов будет знать, какой сервис ей создавать. Значение свойства `useFactory` — функция с двумя аргументами: сама функция-фабрика и ее зависимости (`deps`).

ПРИМЕЧАНИЕ

В листинге 4.6 и множестве других примеров кода из этой книги используются анонимные стрелочные функции (они описаны в приложении А). По сути, анонимная стрелочная функция — более короткая нотация для анонимных функций. Например, конструкция (isDev) => {...} эквивалентна вызову function(isDev) {...}.

Листинг 4.6. Указание поставщика значения среды

```
@Component({
  selector: 'product2',

  providers:[{
    provide: ProductService,
    useFactory: (isDev) => {
      if (isDev){
        return new MockProductService();
      } else{
        return new ProductService();
      }
    },
          deps:["IS_DEV_ENVIRONMENT"]})],
  template: '{{product.title}}'
 }
class Product2Component {...}
...

@NgModule({
  ...
  providers: [ ProductService,
               {provide: "IS_DEV_ENVIRONMENT", useValue:true}]
})
```

Функция-фабрика имеет аргумент isDev, который является зависимостью, внедренной снаружи

Второе свойство, deps, определяет зависимость функции-фабрики (в нашем случае это внедряемое значение IS_DEV_ENVIRONMENT)

Чтобы сделать значение IS_DEV_ENVIRONMENT внедряемым, укажите значения свойств provide и useValue

Поскольку вы внедряете значение в `IS_DEV_ENVIRONMENT` на уровне приложения, любой компонент-потомок, который использует эту фабрику, будет затронут в результате простого перехода значения от `false` к `true`.

Напомним: поставщик соотносит токен с классом или фабрикой, чтобы известить инъектора, как именно создавать объекты. Класс или фабрика могут иметь собственные зависимости, поэтому поставщики должны указать их все. На рис. 4.8 показаны отношения между поставщиками и инъектором из листинга 4.6.

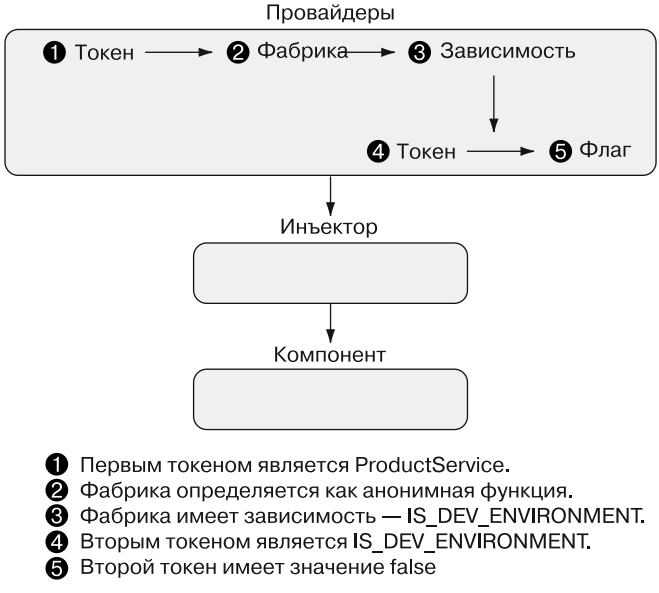


Рис. 4.8. Привязка фабрики к зависимостям

Angular создает дерево поставщиков, находит инъектор и задействует его для элемента `Product2Component`. Angular будет применять инъектор либо компонента, либо его предка. Иерархию инъекторов мы рассмотрим ниже.

4.4.2. Использование OpaqueToken

Внедрение в жестко закодированную строку (такую как `IS_DEV_ENVIRONMENT`) может вызвать проблему, если приложение имеет более одного поставщика, использующего строку с одним и тем же значением для разных целей. Angular предлагает класс `OpaqueToken`, для которого предпочтительно применять в качестве токенов строки.

Представьте, что вам нужно создать компонент, который может получать данные от разных серверов (например, dev, prod и QA). В листинге 4.7 показано, как ввести в код внедряемое значение, `BackendUrl`, — экземпляр класса `OpaqueToken`, а не строку.

Листинг 4.7. Применение OpaqueToken вместо строки

```
import {Component, OpaqueToken, Inject, NgModule} from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { BrowserModule } from '@angular/platform-browser';
export const BackendUrl = new OpaqueToken('BackendUrl');

@Component({
  selector: 'app',
  template: 'URL: {{url}}'
})
```

```
class AppComponent {
  constructor(@Inject(BackendUrl) public url: string) {}
}

@NgModule({
  imports:      [ BrowserModule],
  declarations: [ AppComponent],
  providers: [  {provide:BackendUrl, useValue: 'myQAserver.com'}],
  bootstrap:    [ AppComponent ]
})
class AppModule { }
platformBrowserDynamic().bootstrapModule(AppModule);
```

Вы оборачиваете строку `BackendUrl` в экземпляр класса `OpaqueToken`. Далее в конструкторе этого компонента вместо того, чтобы внедрять значение `vague` типа `string`, внедряете конкретный тип `BACKEND_URL` со значением, предоставленным при объявлении модуля.

4.5. Иерархия инъекторов

Любое Angular-приложение представляет собой дерево вложенных компонентов. Когда веб-страница загружается, Angular создает объект приложения, который имеет свой инъектор. Он, в свою очередь, создает иерархию элементов, имеющих соответственные инъекторы согласно структуре приложения. Например, вам может понадобиться выполнить определенную функцию в момент инициализации приложения:

```
{provide:APP_INITIALIZER, useValue: myappInit}
```

Корневой компонент приложения размещает другие элементы. Если вы включите, например, элемент B в шаблон элемента A, то последний станет предком для первого. Другими словами, корневой компонент является предком для других компонентов-потомков, которые, в свою очередь, могут иметь собственных потомков.

Рассмотрим следующий документ HTML, содержащий корневой компонент, представленный тегом `<app>`:

```
<html>
  <body>
    <app></app>
  </body>
</html>
```

Из следующего кода видно, что `app` является селектором компонента `AppComponent`, который выступает предком для компонентов `<product1>` и `<product2>`:

```
@Component({
  selector: 'app',
  template: `
    <product1></product1>
    <product2></product2>
  `
})
class AppComponent {}
```

Инъектор предка создает инъекторы для каждого потомка, поэтому есть иерархия компонентов и иерархия инъекторов. Кроме того, разметка шаблона каждого компонента может иметь собственные Shadow DOM с элементами и для каждого элемента будет присутствовать собственный инъектор. На рис. 4.9 показана иерархия инъекторов.

Рис. 4.9. Иерархия инъекторов

Когда в коде создается компонент, который требует внедрения определенного объекта, Angular ищет поставщик запрошенного объекта на уровне компонента. При нахождении поставщика используется инъектор компонента. В противном случае Angular проверяет, существует ли поставщик в одном из предков. Если поставщик запрошенного объекта не найден ни на одном уровне иерархии инъекторов, то Angular сгенерирует ошибку.

ПРИМЕЧАНИЕ

Angular создает дополнительный инъектор для «лениво» загружаемых модулей. Поставщики, объявленные в директиве @NgModule модуля, загружаемого «лениво», доступны только на уровне модуля, но не для всего приложения.

В приложении-примере внедряется только сервис, в нем не показывается использование инъекторов элементов. В браузере каждый экземпляр компонента может быть представлен Shadow DOM, содержащим один или несколько элементов в зависимости от того, что определено в шаблоне компонента. Каждый элемент Shadow DOM имеет объект класса `ElementInjector`, который следует той же иерархии «предок — потомок», что и сами элементы DOM.

Допустим, в элемент-компонент HTML `<input>` нужно добавить функцию автозаполнения. Для этого определите директиву следующим образом:

```
@Directive({
  selector: '[autocomplete]'
})
class AutoCompleter {
```

```
  constructor(element: ElementRef) {
    // Здесь будет реализована логика автозаполнения
  }
}
```

Квадратные скобки означают, что автозаполнение можно использовать как атрибут элемента HTML. Ссылка на этот элемент будет автоматически внедрена в конструктор класса `AutoCompleter` с помощью инъектора элементов.

Теперь еще раз взглянем на код из раздела 4.4. Класс `Product2Component` имеет поставщик `MockProductService` на уровне компонента. В классе `Product1Component` не указаны поставщики для типа `ProductService`, поэтому Angular выполнил следующие действия:

- проверил его предка `AppComponent` — поставщик не найден;
- проверил `AppModule` и нашел строку `providers:[ProductService]`;
- использовал инъектор уровня приложения и создал экземпляр класса `ProductService` на уровне приложения.

В случае удаления строки `providers` из класса `Product2Component` и перезапуска приложения оно все еще будет работать, применяя инъектор уровня приложения и один экземпляр класса `ProductService` для обоих компонентов. Если поставщики для одного токена были указаны для предка и потомка и каждый из этих компонентов имел конструктор, который требовал представленный токеном объект, то будут созданы два отдельных экземпляра такого объекта: один для предка и один для потомка.

4.5.1. viewProviders

Для гарантии того, что определенный внедряемый сервис не будет виден потомкам компонента и другим элементам, задействуйте вместо поставщиков свойство `viewProviders`. Предположим, вы пишете многократно используемую библиотеку, применяющую сервис, который не должен быть виден приложениям, работающим с ней. Свойство `viewProviders` поможет сделать сервис закрытым для библиотеки.

Рассмотрим еще один пример. Представьте, что у вас есть следующая иерархия компонентов:

```
<root>
  <product2>
    <luxury-product></luxury-product>
  </product2>
</root>
```

Компоненты `AppModule` и `Product2Component` имеют поставщики, определенные с помощью токена `ProductService`, но второй элемент использует особый класс, который вы должны скрыть от потомков. В этом случае можно применить свойство `viewProviders` для класса `Product2Component`; когда инъектор компонента `LuxuryProductComponent` не найдет поставщика, он поднимется по иерархии. В `Product2Component` он тоже не увидит поставщика и задействует поставщика для `ProductService`, определенного в компоненте `RootComponent`.

ПРИМЕЧАНИЕ

Экземпляр внедряемого объекта создается и разрушается в то же время, когда создается или разрушается компонент, который определяет поставщика для этого объекта.

4.6. Практикум: использование DI для приложения онлайн-аукциона

В главе 3 вы добавили маршрутизацию в приложение-аукцион, и теперь оно может отрисовывать упрощенное представление Product Details (Информация о продукте). В этом упражнении вы реализуете компонент `ProductDetail`, который сможет показывать реальные сведения о продукте.

Главная страница аукциона показана на рис. 4.10. При нажатии одной из ссылок, например `First Product` или `Second Product`, приложение покажет довольно простое представление, которое вы видели на рис. 3.16.

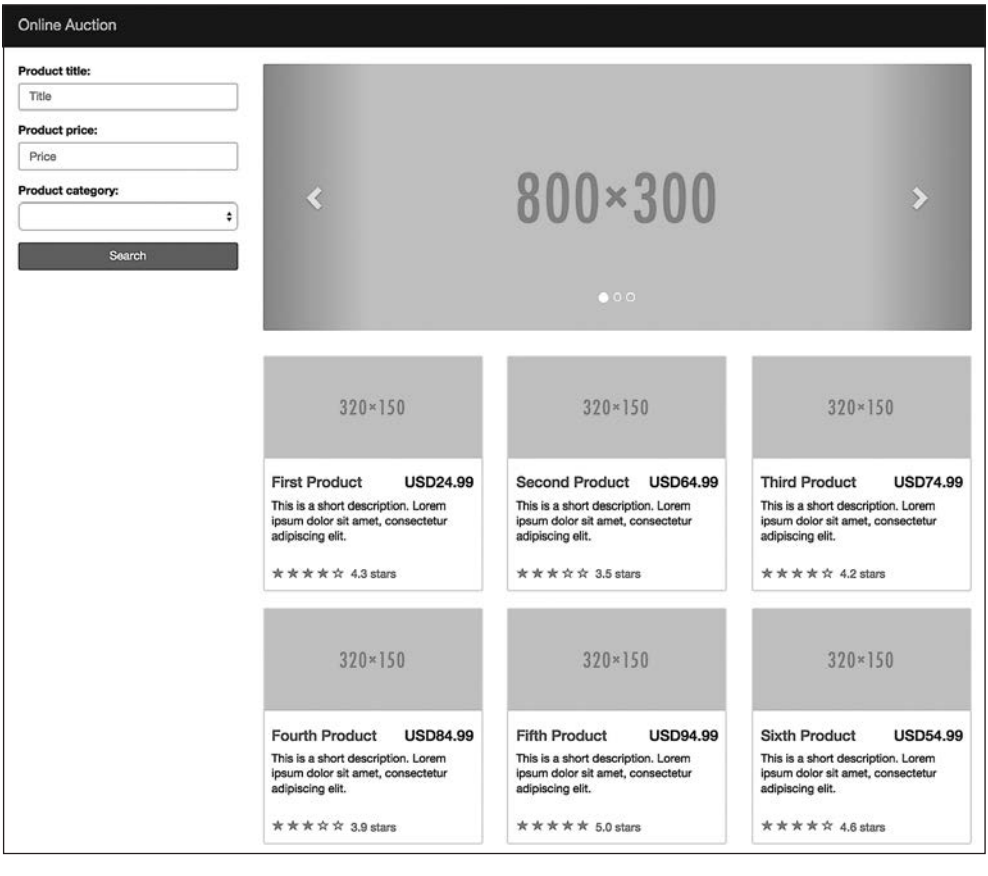


Рис. 4.10. Главная страница аукциона

Ваша цель заключается в том, чтобы отрисовать подробную информацию о выбранном продукте; это позволит продемонстрировать механизм DI в действии. На рис. 4.11 показано, как представление Product Details (Информация о продукте) будет выглядеть по завершении этого упражнения, если выбрать ссылку `First Product`.

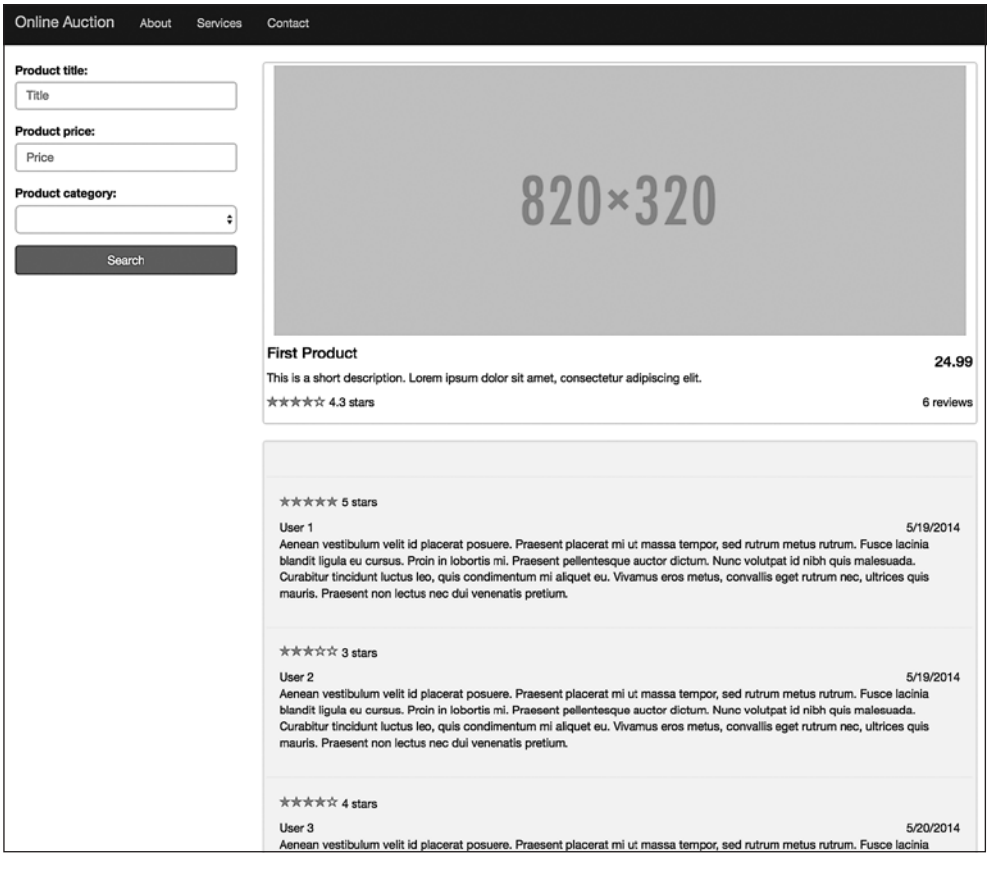


Рис. 4.11. Представление аукциона Product Details

СОВЕТ

> В качестве стартовой точки для этого упражнения мы будем использовать приложение, разработанное в главе 3. Если хотите увидеть итоговый результат проекта, то откройте исходный код, расположенный в каталоге auction для главы 4. В противном случае скопируйте каталог auction из главы 3 в другое место, запустите команду npm install и следуйте инструкциям, представленным в данном разделе.

Теперь, когда вы узнали о поставщиках и внедрении зависимостей, кратко рассмотрим некоторые фрагменты аукциона, созданные в предыдущей главе,

концентрируясь на коде, связанном с внедрением зависимостей. Сценарий из файла app.module.ts определяет поставщики сервиса на уровне приложения, как показано здесь:

```
@NgModule({
    ...
    providers:   [ProductService,
                  {provide: LocationStrategy, useClass: HashLocationStrategy}],
    bootstrap:   [ ApplicationComponent ]
})
export class AppModule { }
```

Поскольку поставщик ProductService определен в модуле, его могут применять все потомки компонента ApplicationComponent. В следующем фрагменте компонента HomeComponent (см. файл home.ts) не указываются поставщики, которые нужно использовать для внедрения ProductService с помощью конструктора — в нем задействован экземпляр класса ProductService, созданный в его предке:

```
@Component({
  selector: 'auction-home-page',
  styleUrls: ['/home.css'],
  template: `...`
})
export default class HomeComponent {
  products: Product[] = [];
  constructor(private productService: ProductService) {
    this.products = this.productService.getProducts();
  }
}
```

Сразу после создания объекта класса HomeComponent в него внедряется сервис ProductService, его метод getProducts() заполняет массив products, привязанный к представлению.

Фрагмент HTML, который отображает содержимое этого массива, использует цикл *ngFor, чтобы отобразить шаблоны <auction-product-item> для каждого элемента массива:

```
<div class="row">
  <div *ngFor="let product of products" class="col-sm-4 col-lg-4 col-md-4">
    <auction-product-item [product]="product"></auction-product-item>
  </div>
</div>
```

Шаблон элемента <auction-product-item> содержит следующую строку:

```
<h4><a [routerLink]="['/products', product.title]">{{ product.title }}</a>
➥ </h4>
```

Нажатие этой ссылки дает маршрутизатору команду отрисовать компонент ProductDetailComponent и предоставляет в качестве параметра маршрута значение свойства product.title. Вы можете модифицировать данный код так, чтобы передавать идентификатор продукта вместо его названия.

Мы кратко разобрали существующий код, чтобы напомнить вам, как запрашивается страница `Product Details`. Теперь реализуем код для создания представления, показанного на рис. 4.11.

4.6.1. Изменение кода для передачи идентификатора продукта в качестве параметра

Откройте файл `product-item.html` и измените строку, содержащую `[routerLink]`, чтобы она выглядела следующим образом:

```
<h4><a [routerLink]="['/products', product.id]">{{ product.title }}</a></h4>
```

Файл `product-item.html` содержит шаблон, используемый для отображения продуктов в представлении Home (Главная страница). Теперь после нажатия названия продукта в маршрут, сконфигурированный для пути `products`, будет отправляться значение идентификатора продукта.

4.6.2. Изменение компонента ProductDetailComponent

Прежде чем начнете писать код, взгляните на рис. 4.12; на нем показано отношение «предок — потомок» между компонентами аукциона. Понимание этих отношений может помочь решить, будут ли использоваться потомками некоторые инъекторы предков.

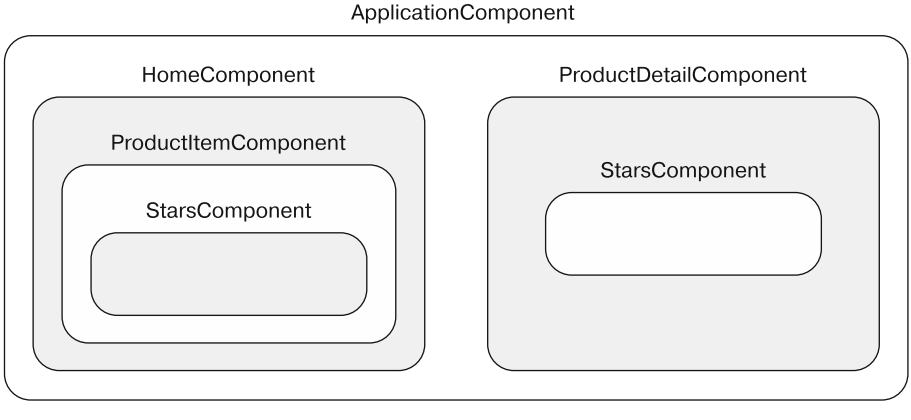


Рис. 4.12. Отношения «предок — потомок» в аукционе

В главе 3 вы внедрили объект типа `ProductService` в компонент `HomeComponent`, но он нужен и в компоненте `ProductDetailComponent`. Можно определить поставщика `ProductService` во время начальной загрузки приложения, чтобы сделать его доступным всем потомкам компонента `ApplicationComponent`. Для этого следуйте инструкциям, представленным ниже.

1. Измените конфигурацию маршрутов в файле `app.module.ts` следующим образом: с `products/:prodTitle` на `products/:productId`. Первые строки декоратора `@NgModule` должны выглядеть так (листинг 4.8).

Листинг 4.8. Изменения в файле app.module.ts

```
@NgModule({
    imports:[ BrowserModule,
            RouterModule.forRoot([
              {path: '', component: HomeComponent},
              {path: 'products/:productId',
                    ➥ component: ProductDetailComponent}
]) ],
```

Поскольку вы передаете идентификатор продукта компоненту `ProductDetailComponent`, его код нужно изменить соответствующим образом.

2. Откройте файл `product-detail.ts` и измените его код так, как показано далее (листинг 4.9).

Листинг 4.9. Изменения в файле product-detail.ts

```
import {Component} from '@angular/core';
import { ActivatedRoute} from '@angular/router';
import {Product, Review, ProductService} from
➥ '../../services/product-service';

@Component({
  selector: 'auction-product-page',
  templateUrl: 'app/components/product-detail/product-detail.html'
})

export default class ProductDetailComponent {
  product: Product;
  reviews: Review[];
  constructor(route: ActivatedRoute, productService: ProductService)
  {
    let prodId: number = parseInt(route.snapshot.params['productId']);
    this.product = productService.getProductById(prodId);
    this.reviews = productService.getReviewsForProduct(this.product.id);
  }
}
```

Angular внедрит объект типа `ProductService` в компонент `ProductDetailComponent`. Когда создается объект класса с аналогичным названием, он вызовет метод `getProductsById()`. Он возвращает один продукт с идентификатором, совпадающим с переданным из представления `Home` (Главная страница) значением `productId`, с помощью аргумента конструктора типа `ActivatedRoute`. Так вы устанавливаете значение переменной `product`.

После этого конструктор вызывает метод `getReviewsForProduct()`, чтобы заполнить массив `reviews`. Вы увидите объявление данного метода, а также класс `Review`, далее в этом разделе.

3. Создайте файл с именем `product-detail.html` в каталоге `product-detail`, который имеет следующее содержимое (листинг 4.10).

Листинг 4.10. Содержимое файла product-detail.html

```
<div class="thumbnail">
    <img src="http://placehold.it/820x320">
    <div>
        <h4 class="pull-right">{{ product.price }}</h4>
        <h4>{{ product.title }}</h4>
        <p>{{ product.description }}</p>
    </div>
    <div class="ratings">
        <p class="pull-right">{{ reviews.length }} reviews</p>
        <p><auction-stars [rating]="product.rating"></auction-stars></p>
    </div>
</div>
<div class="well" id="reviews-anchor">
    <div class="row">
        <div class="col-md-12"></div>
    </div>
    <div class="row" *ngFor="let review of reviews">
        <hr>
        <div class="col-md-12">
            <auction-stars [rating]="review.rating"></auction-stars>
            <span>{{ review.user }}</span>
            <span class="pull-right">
            ➥ {{ review.timestamp | date: 'shortDate' }}</span>
            <p>{{ review.comment }}</p>
        </div>
    </div>
</div>
```

В данном шаблоне HTML применяется локальная привязка для свойств переменной `product`. Обратите внимание на использование квадратных скобок для передачи значения рейтинга в компонент `StarsComponent` (представленный элементом `<auction-stars>`), с которым вы познакомились в главе 2. В этой версии аукциона пользователь может только просматривать отзывы; функциональность Leave a Review (Оставить отзыв) вы реализуете в главе 6.

Оператор канала (`|`) позволяет создавать фильтры, которые могут преобразовывать значения. Выражение `review.timestamp | date: 'shortDate'` принимает временную метку из объекта класса `Review` и отображает ее в формате `shortDate`. Вы можете найти другие форматы дат в документации к Angular, доступной на https://angular.io/api/common/DatePipe. Angular поставляется с несколькими классами, которые могут быть использованы с оператором `pipe`; кроме того, можно создавать собственные фильтры (об этом рассказывается в главе 5). В главе 8 вы увидите, как применять асинхронные `pipe` для автоматического разворачивания ответов сервера.

4. Чтобы сэкономить время, скопируйте в проект файл `app/services/product-service.ts`, который поставляется с кодом приложения аукциона для настоящей главы. В этом файле находятся три класса — `Product`, `Review` и `ProductService`, — а также жестко закодированные данные о продуктах и отзывах. В шаблоне HTML из листинга 4.10 используются следующие классы `Product` и `Review` (листинг 4.11).

Листинг 4.11. Классы Product и Review

```
export class Product {
  constructor(
    public id: number,
    public title: string,
    public price: number,
    public rating: number,
    public description: string,
    public categories: string[]) {
  }
}
export class Review {
  constructor(
    public id: number,
    public productId: number,
    public timestamp: Date,
    public user: string,
    public rating: number,
    public comment: string) {
  }
}
```

Класс `ProductService` показан в листинге 4.12.

Листинг 4.12. Класс ProductService

```
export class ProductService {
  getProducts(): Product[] {
    return products.map(p => new Product(p.id, p.title, p.price, p.rating,
    ➥ p.description, p.categories));
  }
  getProductById(productId: number): Product {
    return products.find(p => p.id === productId);
  }
  getReviewsForProduct(productId: number): Review[] {
    return reviews
      .filter(r => r.productId === productId)
      .map(r => new Review(r.id, r.productId, Date.parse(r.timestamp),
      ➥ r.user, r.rating, r.comment));
  }
}
var products = [
  {
    "id": 0,
    "title": "First Product",
```

```
    "price": 24.99,
    "rating": 4.3,
    "description": "This is a short description. Lorem ipsum dolor sit
    ➥ amet, consectetur adipiscing elit.",
    "categories": ["electronics", "hardware"]},
  {
    "id": 1,
    "title": "Second Product",
    "price": 64.99,
    "rating": 3.5,
    "description": "This is a short description. Lorem ipsum dolor sit
    ➥ amet, consectetur adipiscing elit.",
    "categories": ["books"]}];

var reviews = [
  {
    "id": 0,
    "productId": 0,
    "timestamp": "2014-05-20T02:17:00+00:00",
    "user": "User 1",
    "rating": 5,
    "comment": "Aenean vestibulum velit id placerat posuere. Praesent..."},
  {
    "id": 1,
    "productId": 0,
    "timestamp": "2014-05-20T02:53:00+00:00",
    "user": "User 2",
    "rating": 3,
    "comment": "Aenean vestibulum velit id placerat posuere. Praesent... "
  }];
```

Этот класс имеет три метода: `getProducts()`, возвращающий массив объектов класса `Product`; `getProductById()`, возвращающий один продукт; и `getReviewsForProduct()`, возвращающий массив объектов класса `Review` для выбранного продукта. Все данные о продуктах и обзоры жестко закодированы в массивах `products` и `reviews` соответственно. (Для краткости мы показали лишь фрагменты этих массивов.) Метод `getReviewsForProduct()` фильтрует массив `reviews` для того, чтобы найти отзывы для заданного значения `productId`. Затем он использует функцию `map()` для превращения массива элементов типа `Object` в новый массив объектов типа `Review`.

5. Запустите сервер в каталоге `auction`, введя команду `npm start`. Когда вы увидите главную страницу аукциона, нажмите название продукта, чтобы увидеть представление Product Details (Информация о продукте), показанное на рис. 4.11.

4.7. Резюме

Из этой главы вы узнали, что такое шаблон «Внедрение зависимостей» и как он реализуется в Angular. В приложении-аукционе данный шаблон используется на каждой странице. Вот основные выводы.

- Поставщики регистрируют объекты для будущего внедрения.
- Вы можете создать поставщик не только для объекта, но и для строкового значения.
- Инъекторы формируют иерархию, и если Angular не может найти поставщик для запрошенного типа на уровне компонента, то попробует найти его среди инъекторов предка.
- Значение свойства `providers` видно в элементах-потомках, а свойства `viewProviders` — только на уровне компонентов.

5 Привязки, наблюдаемые объекты и каналы

В этой главе:

- работа с разными привязками данных;
- привязки к атрибутам против привязок к свойствам;
- исследуем наблюдаемые потоки данных;
- рассматриваем события как наблюдаемые потоки данных;
- минимизируем нагрузку на сеть путем отмены нежелательных запросов HTTP;
- минимизируем объем кода с помощью каналов.

Предназначение первых четырех глав данной книги состоит в том, чтобы вы быстро начали разрабатывать приложения с помощью Angular. В этих главах мы обсудили, как применять привязки свойств, обрабатывать события и применять директивы, не вдаваясь в детали.

В настоящей главе мы сделаем небольшую передышку и рассмотрим некоторые из приведенных приемов более подробно. Вы продолжите писать код на TypeScript и увидите пример использования синтаксиса деструктурирования при обработке событий.

5.1. Привязка данных

Привязка данных позволяет соединять данные приложения с пользовательским интерфейсом. Синтаксис привязки снижает объем написанного кода. В главе 2 мы кратко рассмотрели данный синтаксис, и вы использовали его практически в каждом примере предыдущих глав. В частности, видели следующее:

В Angular привязка данных является односторонней. Это означает либо отображение изменений данных в свойствах элемента в интерфейсе, либо привязку событий пользовательского интерфейса к методам компонента. Например, при обновлении свойства компонента `productTitle` представление (шаблон) автоматически обновляется с помощью следующего синтаксиса в шаблоне: `{{productTitle}}`. Аналогично, когда пользователь что-то вводит в поле `<input>`, привязка событий (отмеченное скобками) вызывает обработчик событий с правой стороны знака «равно»:

```
(input) = "onInput()"
```

ПРИМЕЧАНИЕ

В шаблонах двойные фигурные скобки в тексте и квадратные скобки в атрибутах элементов HTML указывают на привязку свойств. Angular связывает интерполированное значение (строку, в которую были внедрены значения выражения) и свойство textContent соответствующего узла DOM. Это присваивание выполняется не один раз — текст постоянно обновляется по мере изменения соответствующего значения.

Что было не так с двухсторонней привязкой в AngularJS?

В AngularJS изменения данных в представлении автоматически обновляют данные, лежащие в его основе (одно направление), что также вызывает обновление данных представления (другое направление). Другими словами, AngularJS использует скрытую от пользователя двухстороннюю привязку данных.

Несмотря на то что двухсторонняя привязка данных в формах упрощает написание кода, применение ее для связывания значений в разных сценариях может значительно снизить производительность крупных приложений. Так происходит вот почему: AngularJS ведет список всех выражений привязки данных на странице, и генерация какого-нибудь события браузера может привести к тому, что AngularJS будет проверять весь список выражений снова и снова до тех пор, пока не убедится в синхронизации всех данных. Во время этого процесса значение одного свойства может обновиться несколько раз.

Несмотря на то что в Angular по умолчанию не используется двухсторонняя привязка данных, вы можете реализовать указанную функциональность. Теперь это ваш выбор, а не создателей фреймворка. В настоящем разделе мы рассмотрим несколько видов привязки данных:

- привязка событий для вызова функций, которые обрабатывают эти события;
- привязка атрибутов для обновления текстовых значений атрибутов элементов HTML;
- привязка свойств для обновления значений свойств элементов DOM;
- привязка шаблонов для преобразования шаблонов представления;
- двухсторонняя привязка данных с помощью `ngModel`.

5.1.1. Привязки к событиям

Чтобы назначить событию функцию-обработчик, нужно поместить имя события в круглых скобках в шаблон компонента. В следующем фрагменте кода показано, как привязать функцию `onClickEvent()` к событию `click`, а функцию `onInputEvent()` — к событию `input`:

```
<button (click)="onClickEvent()">Get Products</button>
<input placeholder="Product name" (input)="onInputEvent()">
```

Когда срабатывает событие, указанное в скобках, значение выражения, расположенного в двойных кавычках, оценивается повторно. В предыдущем примере выражения являются функциями, поэтому вызываются всякий раз при срабатывании соответствующего события.

Если нужно проанализировать свойства объекта события, то добавьте аргумент `$event` в функцию-обработчик. В частности, целевое свойство объекта события представляет собой узел DOM, в котором произошло событие. Объект события будет доступен только в определенной области видимости (то есть в функции — обработчике события). На рис. 5.1 показано, как читать синтаксис привязки событий.

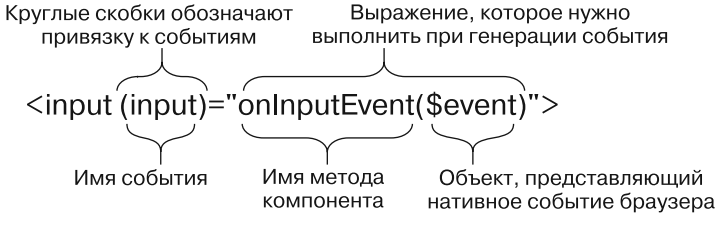


Рис. 5.1. Синтаксис привязки событий

Событие, расположенное в круглых скобках, называется *целью привязки*. Можно привязывать функции к любым стандартным событиям DOM, существующим сегодня (см. раздел Event reference в документации Mozilla Developer Network, https://developer.mozilla.org/en-US/docs/Web/Events), а также к тем, которые будут введены в будущем. Кроме того, можно создавать пользовательские события и привязывать к ним функции-обработчики точно таким же способом (см. «Выходные свойства и пользовательские события» в подразделе 6.1.1).

5.1.2. Привязка к свойствам и атрибутам

Каждый элемент HTML представлен тегом, имеющим *атрибуты*, и браузер создает объект модели DOM, который имеет *свойства* для каждого тега. Пользователь видит такие объекты на экране по мере того, как браузер их отрисовывает. Вы должны понимать, что находится в любой момент времени в трех конкретных областях, таких как:

- документ HTML;
- объект модели DOM;
- отрисованный пользовательский интерфейс.

Документ HTML состоит из элементов, представленных тегами, имеющими атрибуты, которые всегда являются строками. Браузер создает объекты для элементов HTML в виде *объектов модели DOM* (узлов), которые имеют свойства и отрисовываются на веб-странице как пользовательский интерфейс. Когда значения свойств узлов DOM изменяются, страница перерисовывается.

Свойства

Рассмотрим следующий тег `<input>`:

```
<input type="text" value="John" required>
```

Браузер использует эту строку для создания узла в дереве DOM, который является объектом JavaScript типа `HTMLInputElement`. Каждый объект модели DOM имеет API, состоящий из методов и свойств (см. раздел HTMLInputElement в документации Mozilla Developer Network, https://developer.mozilla.org/en-US/docs/Web/API/HTMLInputElement). В частности, объект `HTMLInputElement` содержит свойства `type` и `value` типа `DOMString`, а также свойство `required` типа `Boolean`. Браузер отрисовывает данный узел DOM.

ПРИМЕЧАНИЕ

Браузер синхронизирует отрисованные значения со значениями соответствующих свойств независимо от функциональности синхронизации, предлагаемой каким-либо фреймворком.

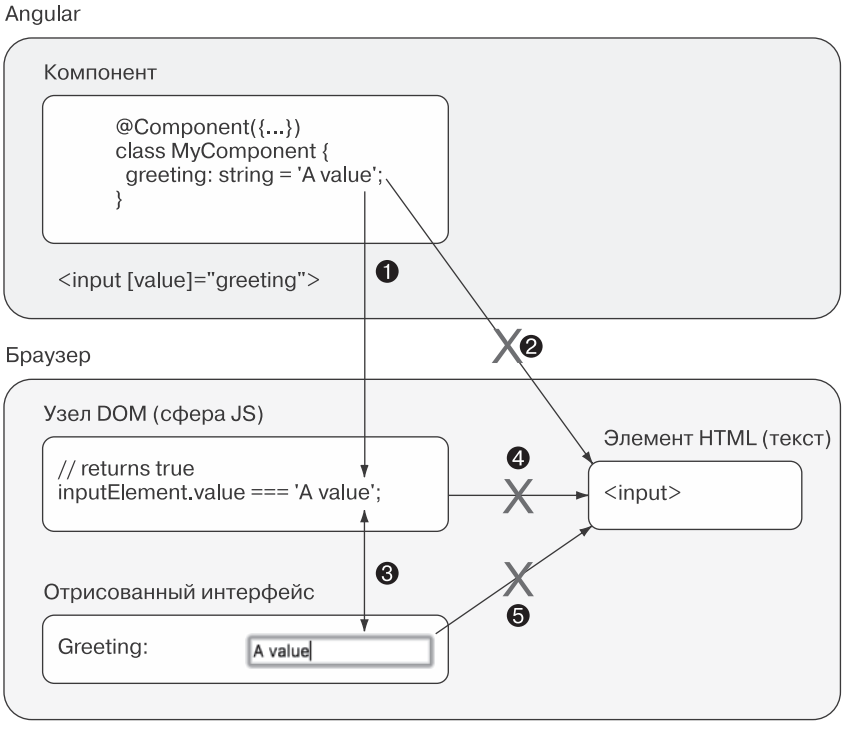
В Angular вы объявляете привязку свойства, поместив его имя в квадратные скобки и присвоив ему выражение (или переменную класса). На рис. 5.2 показано, как работает механизм привязки свойств в Angular. Представьте компонент `MyComponent`, который имеет класс, содержащий переменную `greeting`. Шаблон этого компонента содержит тег `<input>`, классовая переменная `greeting` привязана к свойству `value`.

Компонент приложения может иметь структуру данных, которая служит для него моделью. Код приложения также может изменять свойство модели (например, функцию, что-то вычисляющую, или данные, получаемые с сервера); это заставит отреагировать механизм привязки свойства и приведет к обновлению пользовательского интерфейса.

Когда следует использовать привязку свойств

Привязку свойств целесообразно применять в таких сценариях:

- компонент должен отражать состояние модели в представлении;
- компонент-предок должен обновить свойство своего потомка (см. пункт «Входные свойства» на с. 204).



❶ Angular обновляет DOM с помощью одностороннего связывания свойств (от greeting до свойства объекта DOM). Если сценарий присваивает ссылку на этот элемент <input> переменной inputElement, она будет равна значению A. Обратите внимание на то, что мы используем точечную нотацию для получения доступа к свойству узла.

❷ Механизм связывания свойств в Angular не обновляет атрибут элемента HTML после того, как изменяется значение inputElement.value.

❸ Свойство value узла DOM отображается в пользовательском интерфейсе. Angular обновил узел DOM, а браузер отрисовал новое значение, чтобы DOM и интерфейс были синхронизированы.

❹ Свойство value узла DOM не изменяет атрибут соответствующего элемента HTML.

❺ Браузер не синхронизирует атрибут элемента HTML с интерфейсом, когда пользователь вводит данные в элементе <input>. Пользователь увидит новые значения, которые поступают из модели DOM, а не из документа HTML

Рис. 5.2. Привязка свойств

Атрибуты

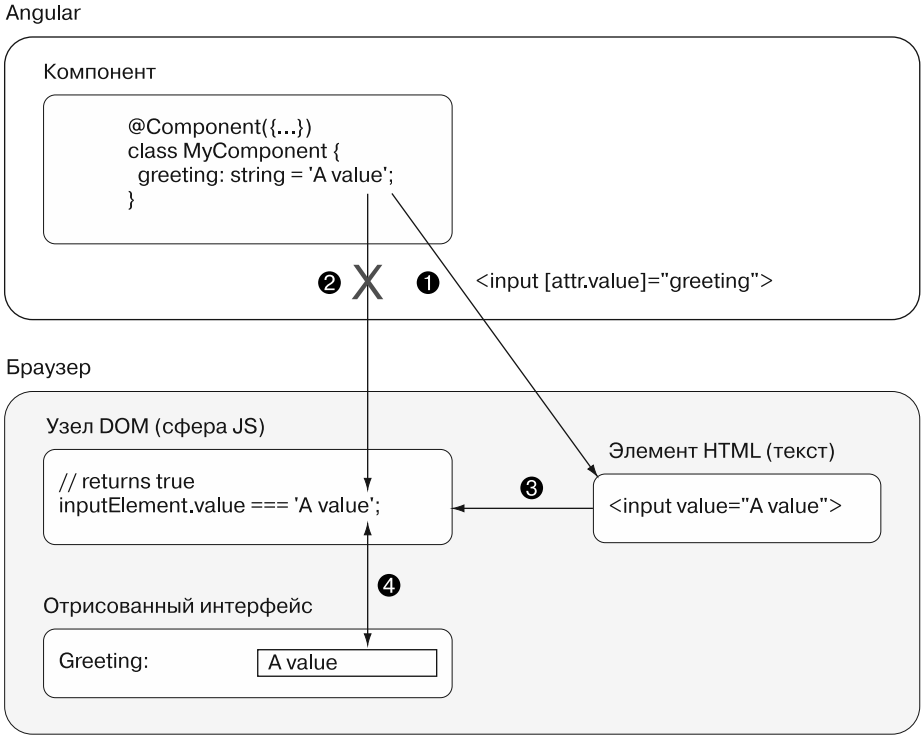
Мы используем слово *«атрибуты»* в контексте документа HTML (но не объекта модели DOM). Привязка атрибутов применяется редко, поскольку браузер для сборки дерева DOM задействует HTML, после чего работает в основном со свойствами объекта модели DOM. Но существуют сценарии, в которых привязка атрибутов

может понадобиться. Например, скрытые атрибуты не поддерживаются в браузере Internet Explorer 10, и он не создаст соответствующий атрибут модели DOM, поэтому, если нужно изменять видимость компонента с помощью стилей CSS, то привязка атрибутов поможет. Еще одним примером является интеграция с фреймворком Google Polymer — это можно сделать, только используя привязку атрибутов.

Как и в случае со свойствами, привязка атрибутов обозначается путем размещения имени атрибута в квадратных скобках. Но для оповещения Angular о том, что нужно привязать атрибут (а не свойство модели DOM), следует добавить префикс `attr.`:

```
<input [attr.value]="greeting">
```

На рис. 5.3 показана привязка атрибутов.



❶ Angular обновляет элемент HTML с помощью одностороннего связывания атрибута (от greeting до атрибута элемента HTML). Обратите внимание на конструкцию attr. в выражении связывания.

❷ Связывание атрибутов в Angular не обновляет узел модели DOM.

❸ В этом случае объект модели DOM получил значение A, поскольку браузер синхронизировал значение атрибута для элемента HTML и модели DOM.

❹ Свойство value узла модели DOM отображается в интерфейсе, поскольку браузер синхронизирует интерфейс и модель DOM

Рис. 5.3. Привязка атрибутов

Посмотрим, как работает привязка свойств и атрибутов, на одном примере. В листинге 5.1 показан элемент `<input>`, который использует привязку атрибута `value` и свойства `value`. Этот фрагмент кода располагается в файле `attribute-vs-property.ts`.

Листинг 5.1. Содержимое файла attribute-vs-property.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule, Component }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

@Component({
  selector: 'app',
  template: `
    <h3>Property vs attribute binding:</h3>
    <input [value]="greeting"
    [attr.value] = "greeting"
    (input)="onInputEvent($event)">
  `
})
class AppComponent {

  greeting: string = 'A value';

  onInputEvent(event: Event): void {
    let inputElement: HTMLInputElement = <HTMLInputElement> event.target;

    console.log(`The input property value = ${inputElement.value}`);
    console.log(`The input attribute value = ${inputElement
    ➥ .getAttribute('value')}`);
    console.log(`The greeting property value = ${this.greeting}`);
  }
}

@NgModule({
  imports: [ BrowserModule],
  declarations: [ AppComponent],
  bootstrap: [ AppComponent ]
})
class AppModule { }

platformBrowserDynamic().bootstrapModule(AppModule);
```

Привязка свойств

Привязка атрибутов

Привязывает стандартное событие ввода данных DOM, которое срабатывает, когда изменяется значение элемента <input>

Обработчик событий onInputEvent() получает объект Event, его целевое свойство содержит ссылку на элемент, в котором генерируется это событие

Вы используете метод getAttribute() для того, чтобы получить значение атрибута, доступ к свойствам можете получить с помощью точечной нотации

Если вы запустите данную программу и начнете набирать текст в поле ввода, то она выведет в консоли браузера содержимое значения `value` модели DOM, атрибута `value` элемента HTML `<input>`, а также содержимое свойства `greeting` компонента `MyComponent`. На рис. 5.4 показана консоль после запуска этой программы и ввода значения `3` в поле `<input>`.

Значение атрибута `value` не изменилось, как и свойства `greeting`; это доказывает, что в Angular не используется двухсторонняя привязка. В AngularJS изменение модели (свойства `greeting`) обновило бы представление, а если бы пользователь изменил данные в представлении, то модель бы автоматически обновилась.

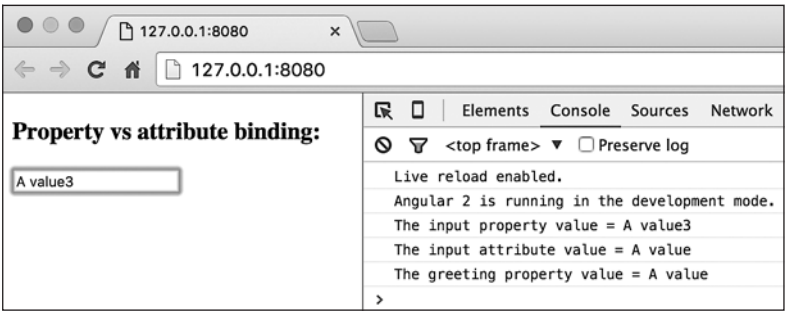


Рис. 5.4. Запуск примера attribute-vs-property

Упрощение кода с помощью деструктурирования

В приложении А мы рассмотрим деструктурирование в рамках ES6, но оно также поддерживается и TypeScript. Деструктурирование может упростить код функции — обработчика событий `onInputEvent()`, показанной в листинге 5.1.

Данная функция получает объект `Event`, а затем с помощью одной строки вы извлекаете значение из целевого свойства. Синтаксис деструктурирования поможет избавиться от строки, извлекающей значение из свойства `event.target`:

```
onInputEvent({target}): void {
  console.log(`The input property value = ${target.value}`);
  console.log(`The input attribute value =
  ➥ ${target.getAttribute('value')}`);
  console.log(`The greeting property value = ${this.greeting}`);
}
```

Использование фигурных скобок в аргументе этой функции указывает ей следующее: «Вы получите объект, который имеет целевое свойство. Просто выдайте мне значение данного свойства».

5.1.3. Привязка в шаблонах

Предположим, вам нужно скрывать или показывать определенный элемент HTML в зависимости от того, выполняется определенное условие или нет. Это можно сделать, привязав флаг типа Boolean к атрибуту `hidden` или отобразив стиль элемента. В зависимости от значения флага данный элемент будет либо показан, либо скрыт, но объект, который представляет искомый элемент, останется в дереве модели DOM.

Angular предлагает использовать *структурные директивы* (`NgIf`, `NgSwitch` и `NgFor`), которые изменяют структуру модели DOM, добавляя или удаляя элементы из нее. Директива `NgIf` в зависимости от выполнения некоего условия может добавить элемент в дерево модели DOM или удалить его из дерева. Директива `NgFor` проходит по массиву и добавляет элементы в дерево DOM для каждого элемента массива. Директива `NgSwitch` добавляет один элемент в дерево модели DOM из на-

бора возможных элементов в зависимости от выполнения определенного условия. С помощью привязки шаблонов можно дать Angular команду сделать это для вас. Лучше удалять элементы, чем скрывать их, если нужно гарантировать, что приложение не будет тратить время на поддержку поведения этих элементов (например, на обработку событий или наблюдение за изменениями).

Шаблоны HTML и директивы Angular

Тег HTML `<template>` (см. раздел <template> в документации к Mozilla Developer Network, https://developer.mozilla.org/en-US/docs/Web/HTML/Element/template) — это необычный тег, поскольку браузер игнорирует его содержимое, если приложение не предоставит сценарий для его анализа и добавления в модель DOM. Angular предлагает так называемый сокращенный синтаксис для директив: они начинаются со звездочки, например `*ngIf` или `*ngFor`. Когда анализатор Angular видит директиву, чье имя начинается со звездочки, он преобразует ее во фрагмент HTML, который использует тег `<template>` и распознается браузерами.

В листинге 5.2 включен один элемент `<span>` и один элемент `<template>`, в нем проиллюстрированы два вида привязки шаблонов с помощью директивы `NgIf`. В зависимости от значения флага (которое изменяется после нажатия кнопки) элементы `<span>` будут добавлены в модель DOM либо убраны из нее.

Листинг 5.2. Содержимое файла template-binding.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule, Component }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

@Component({
    selector: 'app',
    template: `
    <button (click)="flag = !flag">Toggle flag's value</button>
    <p>
      Flag's value: {{flag}}
    </p>
    <p>
      1. span with *ngIf="flag": <span *ngIf="flag">Flag is true</span>
    </p>
    <p>
      2. template with [ngIf]="flag": <template [ngIf]="flag">Flag is true
      ➥ </template>
    </p>
  `
})
class AppComponent {
    flag: boolean = true;
}

@NgModule({
    imports:      [ BrowserModule],
```

```
    declarations: [ AppComponent],
    bootstrap:    [ AppComponent ]
})
class AppModule { }
platformBrowserDynamic().bootstrapModule(AppModule);
```

В отличие от остальных видов привязки, доступных в Angular, привязка шаблона преобразует шаблон представления. Код, показанный в листинге 5.2, добавляет или удаляет значение флага из дерева DOM в зависимости от выполнения определенного условия. Вы будете использовать как сокращенный синтаксис, `*ngIf="flag"`, чтобы обработать элемент `<span>`, так и полную версию, `[ngIf]="flag"`, для обработки содержимого тега `<template>`.

На рис. 5.5 показано, что, когда флаг имеет значение `true`, дерево DOM включает содержимое элементов `<span>` и `<template>`. На рис. 5.6 видно, что, когда флаг имеет значение `false`, дерево DOM не содержит этих элементов.

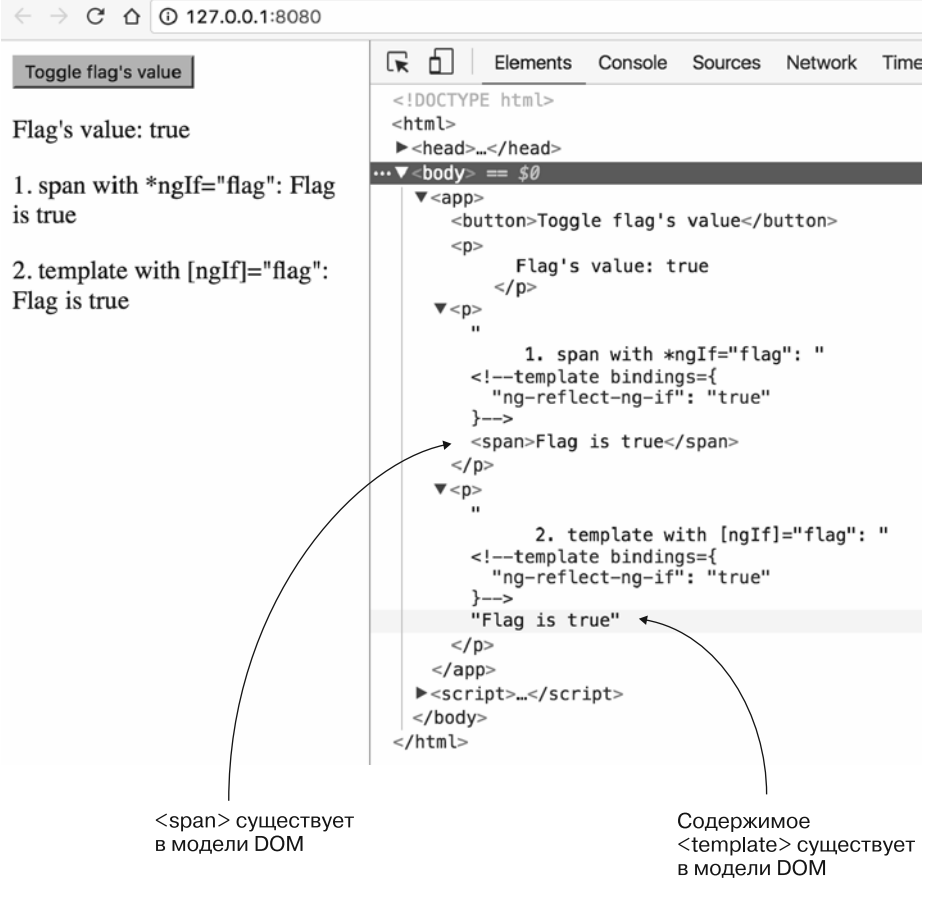


Рис. 5.5. Привязка шаблонов: флаг имеет значение true

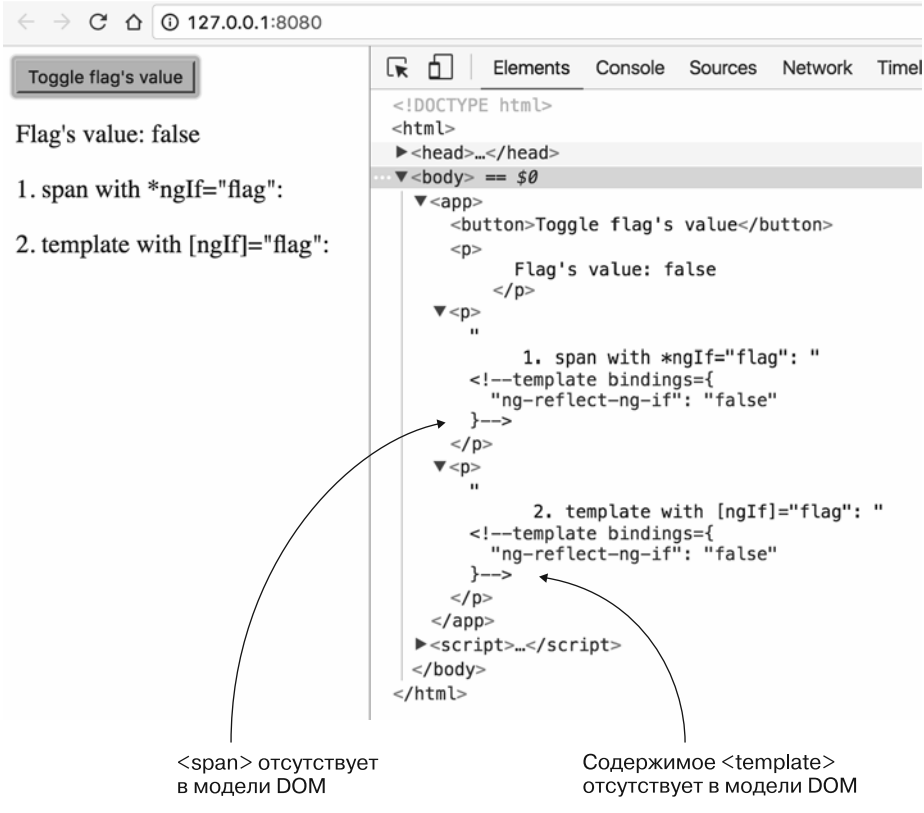


Рис. 5.6. Привязка шаблонов: флаг имеет значение false

Все примеры, приведенные к этому моменту, показывают лишь одностороннюю привязку: либо интерфейса с кодом, либо наоборот. Но существует и другой сценарий, в котором связывание работает в обоих направлениях, мы рассмотрим его далее.

5.1.4. Двухсторонняя привязка данных

Представляет собой простой способ поддержания синхронизации представления и модели. Когда изменяется кто-либо из них, они оба немедленно синхронизируются.

Вы уже знаете, что односторонняя привязка элемента интерфейса и компонента Angular выполняется путем взятия имени события в скобки:

```
<input (input)="onInputEvent($event)">
```

Односторонняя привязка компонента к элементу интерфейса обозначается путем взятия атрибута HTML в квадратные скобки:

```
<input [value]="myComponentProperty" >
```

В некоторых случаях все еще может понадобиться двухсторонняя привязка. Более длинный способ объединения предыдущих двух примеров выглядит так:

```
<input [value]="myComponentProperty"
(input)="onInputEvent($event)>
```

Angular также предлагает более короткую объединенную нотацию: `[()]`. В частности, фреймворк имеет директиву `NgModel`, которую можно применять для настройки двухсторонней привязки (обратите внимание: когда директива `NgModel` используется в шаблонах, ее имя пишется со строчной буквы):

```
<input [(ngModel)] = "myComponentProperty">
```

Вы все еще можете увидеть свойство `myComponentProperty`, но какое событие оно обрабатывает? В данном примере директива `NgModel` применяется вместе с элементом `<input>`. Это событие является триггером по умолчанию для синхронизации изменений, сделанных в пользовательском интерфейсе в элементе HTML `<input>`, с моделью. Но управляющее событие может быть и другим, в зависимости от того, используется ли элемент управления пользовательского интерфейса вместе с `ngModel`. Оно управляется изнутри специальным интерфейсом Angular `ControlValueAccessor`, который выступает в качестве моста между элементом управления и нативным элементом. Интерфейс `ControlValueAccessor` служит для создания пользовательских элементов управления.

Двухсторонняя привязка часто применяется для тех форм, где нужно синхронизировать значения полей форм и свойства модели форм. В главе 7 мы рассмотрим использование директивы `NgModel` более подробно. Вы узнаете, как работать с формами, не задействуя конструкцию `[(ngModel)]` для каждого элемента управления формы; но иногда возникают ситуации, когда это может пригодиться, так что познакомимся с данным синтаксисом.

Предположим, что посадочная страница финансового приложения позволяет пользователю проверять последние цены на акции путем ввода символов в специальное поле. Пользователи зачастую будут вводить символы тех акций, за которыми они следят или которыми обладают, например, AAPL для компании Apple. Можно сохранять последний введенный символ как cookie (или в локальном хранилище HTML5); когда пользователь в следующий раз откроет страницу, программа может считать cookie и заполнить поле ввода. Пользователь должен иметь возможность ввести и другие символы в это поле, и введенные значения должны синхронизироваться с переменной `lastStockSymbol`, играющей роль модели. В листинге 5.3 реализована описанная функциональность.

Листинг 5.3. Содержимое файла two-way-binding.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule, Component } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';
```

```
@Component({
  selector: 'stock-search',
  template: `<input type='text' placeholder= "Enter stock symbol"
  ➥ [(ngModel)] = "lastStockSymbol" />
            <br>The value of lastStockSymbol is {{lastStockSymbol}}`
})

class StockComponent {

    lastStockSymbol: string;

    constructor() {
        setTimeout(() => {
            // Здесь будет находиться код для получения
            // последней введенной акции (пока не реализован)

            this.lastStockSymbol="AAPL";
        }, 2000);
    }
}

@Component({
    selector: 'app',
    directives: [StockComponent],
    template:`<stock-search></stock-search>`

})
class AppComponent {}

@NgModule({
    imports: [ BrowserModule, FormsModule],
    declarations: [ AppComponent],
    bootstrap: [ AppComponent ]
})
class AppModule { }

platformBrowserDynamic().bootstrapModule(AppModule);
```

Дает механизму двухсторонней привязки команду синхронизировать изменения с переменной lastStockSymbol

Переменная lastStockSymbol — это модель, она может измениться в результате действий пользователя, а также программно

Для эмуляции сценария чтения символов последней акции из cookie добавляете задержку в одну секунду, после чего значение переменной lastStockSymbol изменится на AAPL, что отобразится в поле ввода

Импортирует модуль FormsModule, чтобы вы могли использовать директиву NgModel

Переменная `lastStockSymbol` и значение поля `<input>` всегда будут синхронизированы. Вы можете увидеть этот код в действии, запустив сценарий файла `two-way-binding.ts`.

ПРИМЕЧАНИЕ

В листинге 5.3 для реализации двухсторонней привязки данных служит директива NgModel, но для этого можно задействовать и свойства, характерные для самого приложения. Нужно применять в именах этих свойств специальный суффикс, Change. В разделе 6.5 вы увидите, как можно изменять рейтинг продукта, используя двухстороннюю привязку с помощью синтаксиса [(rating)].

В AngularJS двухсторонняя привязка является режимом работы по умолчанию, что кажется простым и элегантным решением для синхронизации представления и модели. Но для сложных интерфейсов, содержащих десятки элементов управления, изменение значения в одном месте может запустить цепную реакцию обновления привязок, от чего пострадает производительность.

Кроме того, двухсторонняя привязка может усложнить отладку, поскольку конкретное значение может измениться по целому ряду причин. Например, это произошло в результате действий пользователя или из-за того, что изменилась другая переменная?

Реализация обнаружения изменений во фреймворке Angular также нетривиальна. При односторонних потоках данных вы всегда знаете, что произошло изменение в определенном элементе интерфейса или свойстве компонента, поскольку только одно свойство кода компонента может изменить заданное значение в интерфейсе.

5.2. Реактивное программирование и наблюдаемые потоки

Принцип реактивного программирования заключается в создании отзывчивых (быстрых) событий, управляемых событиями, в которых наблюдаемый поток событий отправляется подписчикам. В области разработки ПО шаблон «Наблюдатель/Наблюдаемый» распространен довольно широко, он хорошо подходит для любой асинхронной обработки. Но реактивное программирование — не только реализация этого шаблона. Наблюдаемые потоки нельзя отменить, они могут оповестить о конце потока, а данные, отправляемые подписчику, могут быть преобразованы по пути от источника к подписчику с помощью различных операторов (функций).

ПРИМЕЧАНИЕ

> Одной из самых важных характеристик наблюдаемых потоков является тот факт, что они реализуют модель передачи для обработки данных. И наоборот, модель извлечения реализуется путем прохода по массиву с помощью объекта типа Iterable или с использованием функций-генераторов, предоставляемых ES6.

Реактивные расширения реализованы во многих библиотеках, которые поддерживают наблюдаемые потоки. Одной из таких библиотек является RxJS (https://github.com/Reactive-Extensions/RxJS). Она интегрирована в Angular.

5.2.1. Что такое «наблюдаемые потоки» и «наблюдатели»

Наблюдатель — объект, который обрабатывает поток данных, отправляемый *наблюдаемой* функцией. Существует два основных типа наблюдаемых потоков: горячие и холодные. *Холодные* наблюдаемые потоки начинают отправлять данные, только когда какой-то код вызывает функцию `subscribe()`. *Горячие* наблюдаемые потоки отправляют данные даже в том случае, если ни одному подписчику они

не нужны. В этой книге мы будем использовать только холодные наблюдаемые потоки.

Сценарий, который подписывается на наблюдаемый поток, предоставляет объект-наблюдатель, знающий, что делать с элементами потока:

```
let mySubscription: Subscription = someObservable.subscribe(myObserver);
```

Чтобы отменить подписку на поток, вызовите метод `unsubscribe()`:

```
mySubscription.unsubscribe();
```

Наблюдаемым потоком является объект, отправляющий элементы из некоторого источника данных (сокета, массива, событий интерфейса) по одному за раз. Говоря точнее, наблюдаемый поток знает, как делать три вещи:

- отправить следующий элемент;
- сгенерировать ошибку;
- отправить сигнал, оповещающий о том, что поток завершился (был отправлен последний элемент).

Соответственно, объект-наблюдатель предоставляет до трех функций обратного вызова:

- функцию для обработки следующего элемента, отправленного наблюдаемым потоком;
- функцию для обработки ошибок наблюдаемого потока;
- функцию, которая должна быть вызвана, когда поток данных завершается.

ПРИМЕЧАНИЕ

> В приложении А мы рассмотрим объект Promise, способный всего один раз вызывать обработчик событий, указанный в функции then(). Метод subscribe() похож на последовательность вызовов метода then(): по одному вызову на каждый прибывающий элемент данных.

Код приложения может применять последовательность операторов, преобразовывая каждый элемент до того, как тот попадет к функции-обработчику. На рис. 5.7 показан пример перемещения данных от наблюдаемого потока к подписчику (которого реализует наблюдатель). К этим данным применяются два оператора: `map()` и `filter()`. Отправляющая сторона создает оригинальный поток данных (прямоугольники). Оператор `map()` преобразует каждый прямоугольник в треугольник, попадающий к оператору `filter()`, фильтрующему поток так, чтобы только избранные треугольники попадали к подписчику.

Рис. 5.7. От наблюдаемого потока к подписчику

Более реалистичным примером будет поток объектов типа `Customer`, который становится другим потоком, содержащим только свойство `age` для каждого покупателя. Первый поток можно профильтровать так, чтобы оставить только покупателей, чей возраст меньше 50 лет.

ПРИМЕЧАНИЕ

Каждый оператор принимает наблюдаемый объект в качестве аргумента и возвращает также наблюдаемый объект. Это позволяет объединять их в цепочки.

В документации для реактивных расширений (см. раздел Operators в документации к ReactiveX, http://reactivex.io/documentation/operators.html) для иллюстрации операторов используются *интерактивные* схемы. Например, оператор `map()` представлен в виде интерактивной схемы, показанной на рис. 5.8.

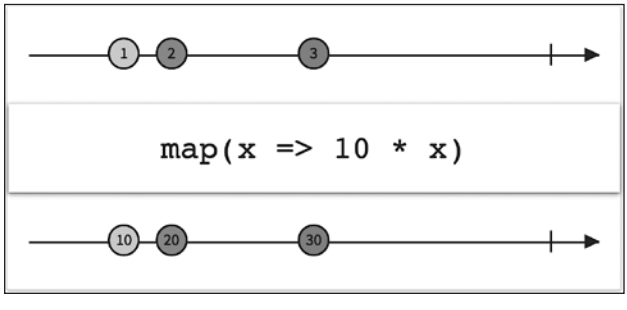


Рис. 5.8. Интерактивная схема для метода map

На этом рисунке показан оператор соотношения, который умножает каждый элемент потока на 10. Вертикальные полоски справа представляют концы соответствующих потоков. В интерактивных схемах ошибки представляются в виде красных крестов.

СОВЕТ

Обратите внимание на сайт RxMarbles (http://rxmarbles.com), где можно найти интерактивные схемы для многих операторов Rx.

От массивов к итерабельным объектам и наблюдаемым потокам

JavaScript имеет несколько полезных методов для работы с массивами данных (представлены ниже).

- Метод `map()` позволяет применять функцию к каждому элементу массива. С помощью `map()` можно преобразовать один массив в другой, не изменяя количество элементов. Например, вызов может выглядеть так: `myArray.map(convertToJSON)`.

- Метод `filter()` позволяет применять функцию к каждому элементу массива, отфильтровывая элементы на основе некоторой бизнес-логики. Например, вызов может выглядеть так: `myArray.filter(priceIsLessThan100)`. Итоговый массив может содержать меньше элементов, чем оригинальный.
- Метод `reduce()` помогает получать среднее значение элементов массива. Например, вызов может выглядеть так: `myArray.reduce((x,y) => x+y)`. Результатом работы метода `reduce()` всегда будет одно значение.

Поток — коллекция данных, передаваемых приложению с течением времени. ES6 вводит концепцию *итерабельных объектов* и *итераторов*, позволяющих работать с массивом как с коллекцией данных и итерировать по его элементам.

Источником итерабельных данных не обязательно должен быть массив. Можно написать функцию-генератор с помощью ES6 (см. приложение А), которая возвращает ссылку на итератор, а затем начать *получать* данные (по одному элементу за раз) из этого итератора, используя следующую конструкцию: `myIterator.next().value`. Для каждого значения можно применить какую-нибудь бизнес-логику, а затем перейти к следующему элементу.

Наблюдаемый объект — более продвинутая версия итератора. Итераторы используют модель извлечения для получения данных, а наблюдаемые объекты отправляют данные подписчикам.

Возможно, концепцию наблюдаемых потоков будет проще понять, если визуализировать данные, поступающие с сервера. Вы увидите подобные примеры и далее в этой главе, и в главе 8, когда научитесь работать с запросами HTTP и WebSockets, но концепцию наблюдаемого потока можно применить и к событиям. Является ли событие одноразовым явлением, которому нужна лишь функция-обработчик? Можно ли рассматривать события как последовательность элементов, поступающую со временем? Далее мы рассмотрим потоки событий.

ПРИМЕЧАНИЕ

О том, как можно превратить любой сервис в наблюдаемый поток, написано в главе 8.

5.2.2. Наблюдаемые потоки событий

Ранее в этой главе вы узнали о синтаксисе привязки событий в шаблонах. Теперь более детально рассмотрим обработку событий.

Каждое событие представлено объектом `Event` (или его потомком), содержащим свойства, описывающие событие. Angular-приложения могут обрабатывать стандартные события модели DOM, а также создавать и *рассылать* пользовательские события.

Функцию — обработчик события можно объявить с необязательным параметром `$event`, содержащим объект JavaScript, чьи свойства описывают событие. В случае

стандартных событий модели DOM можно использовать любые функции или свойства объекта `Event` браузера (см. раздел Event в документации к Mozilla Developer Network, https://developer.mozilla.org/en-US/docs/Web/API/Event).

В некоторых случаях вам не нужно читать свойства объекта события, например, если на странице нажата единственная кнопка, и только это имеет значение. В других ситуациях может понадобиться узнать определенную информацию, например, какой символ был введен в поле `<input>`, когда было отправлено событие `keyup`:

```
template:`<input (keyup)="onKey($event)">`
...
onKey(event:any) {
  console.log("You have entered " + event.target.value);
}
```

В предыдущем фрагменте кода вы получаете доступ к свойству `value` элемента `<input>` путем использования свойства `event.target`, указывающего на элемент, отправленный событию. Но Angular позволяет получить элемент HTML (и его свойства) прямо в шаблоне, объявив *локальную переменную шаблона*, которая всегда будет содержать ссылку на свой элемент HTML.

В следующем фрагменте кода объявлена локальная переменная шаблона `mySearchField` (имя должно начинаться с символа решетки), а также извлекается значение размещающего ее элемента HTML (в нашем случае `<input>`), передающееся функции — обработчику событий вместо ссылки на объект `Event`. Обратите внимание: символ решетки нужен только для того, чтобы объявить локальную переменную в шаблоне; вам не нужно использовать этот символ при работе с данной переменной в коде JavaScript:

```
template:`<input #mySearchField (keyup)="onKey(mySearchField.value)">`
...
onKey(value: string) {
  console.log("You have entered " + value);
}
```

ПРИМЕЧАНИЕ

Если ваш код отправляет пользовательское событие, то оно способно нести характерные для приложения данные, а объект события может быть строго типизирован (а не просто являться типом any). Вы увидите, как это делается, в главе 6 в пункте «Выходные свойства и пользовательские события» на с. 26.

Традиционное приложение JavaScript рассматривает отправленное событие как одноразовое явление; например, один щелчок приводит к одному вызову функции. Angular предлагает другой подход, при котором события считаются наблюдаемыми потоками данных, приходящими время от времени. Обработка наблюдаемых потоков — важный прием; им следует овладеть, поэтому рассмотрим его подробнее.

Подписываясь на поток, ваш код выражает заинтересованность в получении его элементов. Во время подписки вы указываете код, который нужно вызвать при

получении следующего элемента, а также — опционально — код для обработки ошибок и сигнала о завершении потока. Зачастую вы укажете несколько операторов, а затем вызовете метод `subscribe()`.

Как можно применить это к событиям, поступающим от интерфейса? Например, использовать привязку событий, которая обрабатывает несколько событий `keyup` и значение `lastStockSymbol`:

```
<input type='text' (keyup) = "getStockPrice($event)">
```

Достаточно ли хороша данная техника для обработки нескольких событий? Представьте, что предыдущий код используется для получения коммерческого предложения на акции AAPL. После того как пользователь введет первый символ A, функция `getStockPrice()` создаст основанный на промисе запрос на сервер, который вернет цену на акции A, если таковые имеются. Далее пользователь вводит второй символ A, что приводит к отправке еще одного запроса на сервер, на этот раз для получения предложения с расценками для AA. Процесс повторяется для комбинаций *AAP* и *AAPL*.

Вы хотите другого, поэтому создаете задержку 500 миллисекунд, чтобы дать пользователю достаточно времени на ввод нескольких букв. Здесь поможет функция `setTimeout()`!

Что, если пользователь набирает текст медленно и за 500 миллисекунд успеет набрать только AAP? На сервер отправится первый запрос для данной комбинации, а спустя 500 миллисекунд будет отправлен второй запрос для комбинации AAPL. Программа не может отменить первый HTTP-запрос в случае возврата сервером объекта `Promise`, поэтому вам придется скрестить пальцы, понадеявшись, что пользователи будут набирать текст быстро и не перегрузят сервер ненужными запросами.

С помощью наблюдаемых потоков можно решить данную проблему более удачно, отдельные компоненты Angular могут их генерировать. Например, класс `FormControl` является одним из основных классов для обработки форм, он представляет собой элементы формы. Каждый такой элемент имеет собственный объект класса `FormControl`. По умолчанию, когда изменяется значение элемента формы, класс `FormControl` генерирует событие `valueChanges`, создающее наблюдаемый поток, на который можно подписаться.

Напишем небольшое приложение, использующее простую форму с одним полем ввода, генерирующим наблюдаемый поток. Чтобы понять следующий пример, нужно знать, что элементы формы связаны со свойствами компонента Angular с помощью атрибута `formControl`.

ПРИМЕЧАНИЕ

Существует способ программирования форм с использованием директив, размещаемых в шаблоне компонента — они называются шаблон-ориентированными формами. Кроме того, можно программировать формы путем создания объектов, связанных с формами, в коде TypeScript ваших компонентов. Такие формы называются реактивными. Мы подробнее рассмотрим формы в главе 7.

В листинге 5.4 перед вызовом метода `subscribe()` применяется всего один оператор, `debounceTime()`. RxJS поддерживает десятки операторов, пригодных к использованию для наблюдаемых потоков (см. документацию к RxJS, https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/observable.md), но в Angular реализованы не все из них. Именно поэтому нужно импортировать дополнительные операторы из библиотеки RxJS, которая представляет собой зависимость Angular. Оператор `debounceTime()` позволяет указать задержку генерации элементов данных потока.

Листинг 5.4. Содержимое файла observable-events.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule, Component }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormControl,  ReactiveFormsModule} from '@angular/forms';
import 'rxjs/add/operator/debounceTime';

@Component({
    selector: "app",
    template: `
        <h2>Observable events demo</h2>
      <input type="text"
       placeholder="Enter stock" [formControl]="searchInput">
    `
})
class AppComponent {

    searchInput: FormControl = new FormControl('');

    constructor(){

        this.searchInput.valueChanges
            .debounceTime(500)
            .subscribe(stock => this.getStockQuoteFromServer(stock));
        }

        getStockQuoteFromServer(stock: string) {

            console.log(`The price of ${stock} is ${100*Math.random()
            ➥ .toFixed(4)}`);
        }
}

@NgModule({
    imports: [ BrowserModule, ReactiveFormsModule],
    declarations: [ AppComponent],
    bootstrap: [ AppComponent ]
})
class AppModule { }

platformBrowserDynamic().bootstrapModule(AppModule);
```

Вы можете импортировать либо реализацию определенных операторов, как это сделано здесь, либо их все с помощью команды import 'rxjs/Rx';

Этот элемент <input> представлен ngFormControl с именем search

Ждем 500 миллисекунд перед отправкой следующего события, несущего содержимое элемента <input>

Подписывается на наблюдаемый поток

Импортирует модуль, поддерживающий реактивные формы

Метод `subscribe()` создает объект класса `Observer`, который в нашем случае передает каждое значение из потока, сгенерированное элементом `searchInput`, методу `getStockQuoteFromServer()`. В реальном сценарии этот метод создаст запрос на сервер, и вы увидите такое приложение в следующем разделе; сейчас же данная функция генерирует случайное число.

Если вы не использовали оператор `debounceTime()`, то событие `valueChanges` будет отправляться после каждого символа, введенного пользователем. Для предотвращения обработки каждого нажатия клавиши вы даете элементу `searchInput` команду отправлять данные с задержкой 500 миллисекунд, что позволяет пользователю ввести несколько символов до того, как содержимое поля будет отправлено в поток. На рис. 5.9 показан снимок экрана, сделанный после того, как мы запустили приложение и ввели комбинацию символов AAPL в поле ввода.

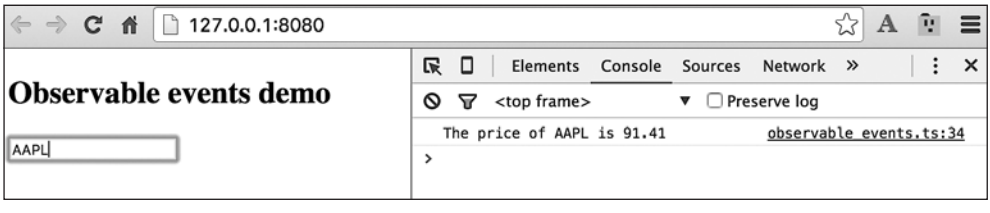


Рис. 5.9. Получение цены для комбинации символов AAPL

СОВЕТ

Независимо от того, сколько операторов вы введете одновременно, ни один из них не будет вызван до тех пор, пока вы не вызовете метод subscribe().

ПРИМЕЧАНИЕ

В листинге 5.4 обрабатывается наблюдаемый поток, который предоставляется объектом FormControl, когда объект модели DOM отправляет событие change. Если вы предпочитаете генерировать наблюдаемый поток на основе другого события (например, keyup), то можете использовать API Observable.fromEvent() из библиотеки RxJS (см. документацию к RxJS на GitHub, https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/operators/fromevent.md).

Вы можете возразить, что есть способ реализовать предыдущий пример по-другому: обрабатывая событие `input`, отправляемое, когда пользователь заканчивает вводить данные в поле и выводит его из фокуса. Это справедливо, но существует множество сценариев, при которых вам необходим мгновенный ответ от сервера, например, нужно получить и профильтровать коллекцию данных по мере ввода значений пользователем.

В листинге 5.4 не выполняются сетевые запросы на сервер для получения цен на акции — вы генерируете случайные числа на компьютере пользователя. Даже если пользователь введет неверную комбинацию символов, то это приведет

к вызову метода `Math.random()`, что незначительно скажется на производительности приложения. В реальных приложениях опечатки, сделанные пользователем, могут сгенерировать сетевые запросы, вызывающие задержки из-за возврата сервером предложений для неверно введенных символов. В следующем разделе мы покажем, как отменять ожидающие серверные запросы для наблюдаемых потоков.

5.2.3. Отменяем наблюдаемые потоки

Одним из преимуществ наблюдаемых потоков перед промисами является тот факт, что их можно отменить. В предыдущем разделе мы показали сценарий, в котором опечатка может привести к созданию бесполезных запросов на сервер. Реализация представления Master-detail (Один ко многим) — еще один пример, когда может понадобиться отменить запрос. Предположим, пользователь нажимает строку в списке товаров для получения подробной информации о продукте, которую нужно получить с сервера. Затем он передумывает и нажимает другую строку, что приводит к созданию другого запроса; в этом случае ожидающий запрос в идеале должен быть отменен.

Рассмотрим способы отмены ожидающих запросов путем создания приложения, отправляющего HTTP-запросы по мере того, как пользователь вводит данные в поле ввода. Нужно обработать два наблюдаемых потока:

- наблюдаемый поток, создаваемый полем поиска;
- наблюдаемый поток, создаваемый HTTP-запросами, отправленными, когда пользователь вводит данные в поле поиска.

В этом примере (файл `observable-events-http.ts`) вы будете использовать бесплатный погодный сервис http://openweathermap.org, которая предоставляет API, позволяющий делать запросы о погоде в городах по всему миру. Она возвращает информацию о погоде в формате JSON. Например, чтобы получить текущую температуру в Лондоне, измеренную в градусах Фаренгейта (`units=imperial`), используйте следующий URL:

```
http://api.openweathermap.org/data/2.5/
➥ find?q=London&units=imperial&appid=12345
```

Чтобы задействовать этот сервис, перейдите по ссылке openweathermap.org и получите идентификатор приложения (application ID, appid). Код, показанный в листинге 5.5, создает URL путем конкатенации базового URL, введенного имени города и идентификатора приложения. Когда пользователь вводит имя города, код подписывается на поток событий и отправляет запросы HTTP. Если новый запрос отправляется до того, как приложение получает результат предыдущего запроса, то оператор `switchMap()` отменяет предыдущий запрос и отправляет новый. Отмену ожидающих запросов нельзя выполнить, применяя ожидания. В этом примере также используется директива `FormControl`, с ее помощью генерируется наблюдаемый поток для поля ввода, где пользователь набирает название города.

Листинг 5.5. Содержимое файла observable-events-http.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule, Component } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormControl, ReactiveFormsModule} from '@angular/forms';
import {HttpModule, Http} from '@angular/http';

import {Observable} from 'rxjs/Rx';
import 'rxjs/add/operator/switchMap';
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/debounceTime';

@Component({
    selector: "app",
    template: `
      <h2>Observable weather</h2>
      <input type="text" placeholder="Enter city" [formControl]=
       "searchInput">
      <h3>{{temperature}}</h3>
    `
})
class AppComponent {
  private baseWeatherURL: string=
➥ 'http://api.openweathermap.org/data/2.5/find?q=';
  private urlSuffix: string =
  ➥ "&units=imperial&appid=ca3f6d6ca3973a518834983d0b318f73";

  searchInput: FormControl = new FormControl('');
  temperature: string;

  constructor(private http:Http){

      this.searchInput.valueChanges
        .debounceTime(200)
        .switchMap(city => this.getWeather(city))
        .subscribe(

            res => {
                if (res['cod'] === '404') return;
                if (!res.main) {
                    this.temperature ='City is not found';
                } else {

                    this.temperature =
                       `Current temperature is ${res.main.temp}F, ` +
                        `humidity: ${res.main.humidity}%`;
                }
            },
            err => console.log(`Can't get weather. Error code: %s, URL: %s`,
            ➥ err.message, err.url),
```

Импортирует требуемую поддержку HTTP

Оператор switchMap() принимает введенное значение из поля ввода (первый наблюдаемый поток) и передает его методу getWeather(), отправляющему HTTP-запрос погодному сервису

Метод subscribe() необходим для того, чтобы наблюдаемый поток начал отправлять данные, в нашем случае с интервалом 200 миллисекунд

Второй аргумент метода subscribe() — это функция обратного вызова, которая вызывается при ошибке

```
            () => console.log(`Weather is retrieved`)
        );
  }

  getWeather(city: string): Observable<Array<string>> {
    return this.http.get(this.baseWeatherURL + city + this.urlSuffix)

        .map(res => {
            console.log(res);
            return res.json()});
  }
}

@NgModule({
    imports: [ BrowserModule, ReactiveFormsModule,
               HttpModule],
    declarations: [ AppComponent],
    bootstrap: [ AppComponent ]
})
class AppModule { }

platformBrowserDynamic().bootstrapModule(AppModule);
```

Третий аргумент метода subscribe() вызывается после того, как поток завершится

Метод getWeather() создает URL и определяет HTTP-запрос GET

Оператор map() принимает данные, которые поступают в формате JSON, обернутые в объект ответа, и преобразовывает их в объект

Добавляет модуль HttpModule

Обратите внимание на два наблюдаемых потока в листинге 5.5:

- директива `FormControl` создает наблюдаемый поток для событий поля ввода (`this.searchInput.valueChanges`);
- метод `getWeather()` тоже возвращает наблюдаемый поток.

Вы используете оператор `switchMap()` вместо `subscribe`, когда функция, обрабатывающая сгенерированные наблюдаемым потоком данные, также может возвращать наблюдаемый поток. Далее вы применяете метод `subscribe()` для второго наблюдаемого потока:

```
Observable1 ➔ switchMap(function) ➔ Observable2 ➔ subscribe()
```

Вы переключаетесь с первого наблюдаемого потока на второй. Если поток `Observable1` отправит новое значение, но функция, которая создает `Observable2`, еще не закончила свою работу, завершается принудительно. Метод `switchMap()` отписывается, подписывается заново на `Observable1` и начинает обрабатывать новое значение из этого потока.

Что, если наблюдаемый поток, поступающий из интерфейса, отправляет новое значение до того, как метод `getWeather()` вернет наблюдаемое значение? В этом случае метод `switchMap()` завершает запущенный вызов `getWeather()`, получает новое значение для названия города, пришедшего из интерфейса, и снова вызывает метод `getWeather()`. При завершении вызова `getWeather()` он также отменяет HTTP-запрос, который выполнялся медленно и не успел закончиться в срок.

Первый аргумент метода `subscribe()` содержит метод обратного вызова для обработки данных, поступающих с сервера. Код в выражении со стрелкой специфичен для API, предоставленного сервисом погоды. Вы должны извлечь значение температуры и влажности из возвращенной строки в формате JSON. API,

предлагаемый именно этим сервисом, сохраняет коды ошибок в ответе, так что вы вручную обрабатываете статус 404, не прибегая к использованию функции обратного вызова.

Теперь убедимся в отмене предыдущего запроса. Для ввода слова London нужно больше чем 200 миллисекунд, указанных для метода `debounceTime()`. Это значит, что событие `valueChanges` отправит наблюдаемые данные больше одного раза. Для гарантии того, что запрос на сервер длится дольше 200 миллисекунд, вам нужно медленное соединение с Интернетом.

ПРИМЕЧАНИЕ

В конструкторе, показанном в листинге 5.5, очень много кода, что может показаться тревожным звоночком для тех разработчиков, кто предпочитает использовать конструкторы только для инициализации переменных, не запуская в них код, для выполнения которого требуется продолжительное время. Однако если вы взглянете внимательно, то обнаружите, что в конструкторе мы всего лишь подписываемся на два наблюдаемых потока (события интерфейса и HTTP-сервис). Мы не обрабатываем данные до тех пор, пока пользователь не начинает вводить название города, и это происходит уже после того, как компонент отрисован.

Мы запустили предыдущий пример, а затем включили регулирование на панели Developer Tools (Инструменты разработчика) браузера Chrome, чтобы эмулировать медленное соединение GPRS. В результате при вводе слова *London* было сгенерировано четыре вызова метода `getWeather()` для сочетаний *Lo*, *Lon*, *Lond* и *London*. Соответственно мы отправили четыре HTTP-запроса, и три из них были автоматически отменены оператором `switchMap()`, как показано на рис. 5.10.

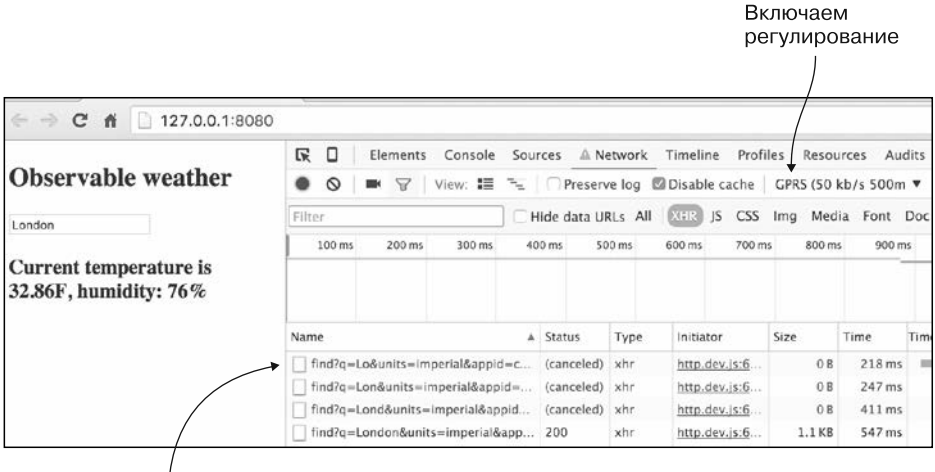


Рис. 5.10. Запуск файла observable_events_http.ts

Написав всего несколько строк кода, вы сэкономили пропускную способность, избавившись от необходимости отправлять четыре HTTP-запроса для городов, не интересных вам и даже, возможно, несуществующих. Как было сказано в главе 1, хорошим считается тот фреймворк, который позволяет писать меньше кода.

Фильтры — еще одна особенность Angular, позволяющая делать больше и при этом писать меньше кода.

ПРИМЕЧАНИЕ

В главе 9 содержится переработанная версия погодного приложения.

5.3. Каналы

Канал — это элемент шаблона, позволяющий преобразовывать значение в желаемый вид. Указывается путем добавления вертикальной полоски (|) и имени канала сразу после значения, которое нужно преобразовать:

```
template: `<p>Your birthday is {{ birthday | date }}</p>`
```

Angular поставляется с несколькими заранее определенными каналами, каждый из них имеет класс, в котором реализована его функциональность (например, `DatePipe`), а также имя, пригодное к использованию в шаблоне (например, `date`):

- `UpperCasePipe` позволяет преобразовывать входную строку в такую же строку в верхнем регистре, применяя конструкцию `| uppercase`, размещаемую в шаблоне;
- `DatePipe` дает возможность отображать дату в разных форматах с помощью конструкции `| date`;
- `CurrencyPipe` преобразует число в желаемую валюту, задействуя конструкцию `| currency`;
- `AsyncPipe` поможет извлечь данные из обертки, роль которой выполняет наблюдаемый поток, с помощью конструкции `| async`. Вы увидите пример кода, использующего эту особенность, в главе 8.

Для некоторых каналов входные параметры не нужны (например, `uppercase`), для других же — наоборот (например, `date:'medium'`). Можно объединить в цепочку столько каналов, сколько необходимо. В следующем фрагменте кода показывается, как отобразить значение переменной `birthday` в формате `medium date` и верхнем регистре (например, `JUN 15, 2001, 9:43:11 PM`):

```
template=
    `<p>
        {{ birthday | date:'medium' | uppercase}}
    </p>`
```

Как видите, можно преобразовать данные в желаемый формат и показать в верхнем регистре, не прилагая особых усилий (форматы дат описаны в документации к Angular `DatePipe`, https://angular.io/api/common/DatePipe).

Обходное решение для неработающих каналов

На момент написания этой книги каналы `date`, `number` и `currency` не работают во всех браузерах. Решить данную проблему можно двумя путями.

- Добавить сервис `polyfill` в ваш файл `index.html`:

```
<script src="https://cdn.polyfill.io/v2/
➥ polyfill.min.js?features=Intl.~locale.en"></script>
```

Сервис выполнит полизаполнение для всего, что нужно вашему браузеру.

- Если не хотите загружать сценарии из CDN (или это запрещено), то добавьте в ваш проект пакет `internationalization`:

```
npm install intl@1.1.0 --save
```

Затем добавьте следующие строки в ваш файл `index.html`:

```
<script src="node_modules/intl/dist/Intl.min.js"></script>
<script src="node_modules/intl/locale-data/jsonp/en.js"></script>
```

Второе решение увеличит размер вашего приложения на 33 Кбайт.

О каналах можно подробнее прочитать в документации к Angular на https://angular.io/guide/pipes, которая включает имя класса, реализующего определенный канал, а также примеры его использования.

5.3.1. Пользовательские каналы

Помимо заранее определенных каналов, Angular предлагает простой способ создания пользовательских каналов, которые могут содержать код, характерный именно для вашего приложения. Нужно создать класс с аннотацией `@Pipe`, реализующий интерфейс `PipeTransform`. Он имеет следующую сигнатуру:

```
export interface PipeTransform {
  transform(value: any, ...args: any[]): any;
}
```

Это говорит о том, что пользовательский класс канала должен реализовать всего один метод, имеющий показанную сигнатуру. Первый параметр `transform` принимает преобразуемое значение, а второй определяет ноль или больше параметров, необходимых вашему алгоритму преобразования. Имя фильтра, используемое в шаблоне, следует указать в аннотации `@Pipe`. Если ваш компонент применяет пользовательские каналы, то их нужно явно указать в свойстве `pipes` аннотации `@Component`.

В предыдущем разделе пример с погодой показал температуру в Лондоне, измеренную в градусах Фаренгейта. Но в большинстве стран используется метрическая система и температура измеряется в градусах Цельсия. Напишем пользовательский канал, который преобразует температуру из градусов Фаренгейта в градусы Цельсия и наоборот. Код нашего канала (листинг 5.6) может быть применен в шаблоне под именем `temperature`.

Листинг 5.6. Содержимое файла temperature-pipe.ts

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

@Pipe({name: 'temperature'})
export class TemperaturePipe implements PipeTransform {

    transform(value: any[], fromTo: string): any {

      if (!fromTo) {
        throw "Temperature pipe requires parameter FtoC or CtoF ";
      }

      return (fromTo == 'FtoC') ?
                (value - 32) * 5.0/9.0: // F to C
                value * 9.0 / 5.0 + 32; // C to F
    }
}
```

Канал носит имя temperature, он может быть использован в шаблоне компонента

Пользовательский канал реализует интерфейс PipeTransform, поэтому вы должны добавить метод transform

Если этот канал используется без предоставления формата, то вы должны сгенерировать ошибку. В качестве альтернативы можете вернуть предоставленное значение без преобразования

Далее показан код компонента (файл `pipe-tester.ts`), который использует канал `temperature` (листинг 5.7). Изначально эта программа будет преобразовывать температуру из градусов Фаренгейта в градусы Цельсия (формат `FtoC`). После нажатия кнопки вы можете изменить направление преобразования температуры.

Листинг 5.7. Содержимое файла pipe-tester.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule, Component }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';
import {TemperaturePipe} from './temperature-pipe';

@Component({
  selector: 'app',
  template:`<input type='text' value="0"
            placeholder= "Enter temperature" [(ngModel)] = "temp">
            <button (click)="toggleFormat()">Toggle Format</button>
```

```
            <br>In {{targetFormat}} this temperature is {{temp | temperature:
            ➥  format | number:'1.1-2'}}`
})

class AppComponent {

    temp: number;
    toCelsius: boolean=true;
    targetFormat: string ='Celsius';
    format: string='FtoC';

    toggleFormat(){

        this.toCelsius = !this.toCelsius;
        this.format = this.toCelsius? 'FtoC': 'CtoF';

        this.targetFormat = this.toCelsius?'Celsius':'Fahrenheit';
    }
}

@NgModule({
    imports:      [ BrowserModule, FormsModule],
    declarations: [ AppComponent, TemperaturePipe],
    bootstrap:    [ AppComponent ]
})
class AppModule { }

platformBrowserDynamic().bootstrapModule(AppModule);
```

Объединяет канал temperature с каналом number от Angular, чтобы отобразить полученное значение температуры как число, которое имеет как минимум один разряд перед запятой и как минимум две цифры после нее

Изначальное значение, FtoC, передается как параметр в канал temperature

Когда пользователь нажимает кнопку, направление преобразования переключается, выводимый текст меняется соответствующим образом

Импортирует модуль FormsModule, что необходимо для поддержки ngModel

Явно указывает пользовательский канал в объявлении модуля

В следующем разделе вы создадите еще один пользовательский фильтр, чтобы использовать его для продуктов аукциона.

5.4. Практикум: фильтрация продуктов онлайн-аукциона

В этом упражнении вы будете использовать наблюдаемые потоки событий, чтобы фильтровать предлагаемые продукты на главной странице онлайн-аукциона. Аукцион (или любой онлайн-магазин) может отображать множество продуктов, что затрудняет поиск необходимого товара.

Например, пользователи могут запомнить всего несколько букв из названия продукта. Чтобы им не пришлось просматривать целые страницы, заполненные продуктами, вы позволяете им ввести название продукта для отсева тех, которые им не подходят. Важно и то, что отрисовываемый список продуктов должен изменяться по мере ввода названия пользователем.

Здесь можно воспользоваться реактивными наблюдаемыми потоками событий. Пользователь вводит букву, что отправляет следующий элемент потока, представляющий содержимое поля поиска. Подписчик этого потока немедленно выполняет

фильтрацию и повторно отрисовывает продукты в интерфейсе. В таком сценарии вы не делаете никаких запросов на сервер.

ПРИМЕЧАНИЕ

В качестве стартовой точки для этого упражнения мы будем использовать ту версию онлайн-аукциона, что была разработана в главе 4. Если хотите увидеть финальную версию проекта, то можете просмотреть исходный код, который представлен в каталоге auction для главы 5. В противном случае скопируйте каталог auction из главы 4 в отдельное место, запустите команду npm install и действуйте в соответствии с инструкциями, приведенными в данном разделе.

Следуйте шагам, описанным ниже.

1. Создайте пользовательский канал, `FilterPipe`. Для этого создайте новый подкаталог `pipes`, в который нужно поместить следующий файл `filter-pipe.ts`, реализуя пользовательский канал `FilterPipe` (листинг 5.8).

Листинг 5.8. Содержимое файла filter-pipe.ts

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

@Pipe({name: 'filter'})
export class FilterPipe implements PipeTransform {

  transform(list: any[], filterByField: string, filterValue: string): any {

    if (!filterByField || !filterValue) {
        return list;
    }

    return list.filter(item => {
        const field = item[filterByField].toLowerCase();
        const filter = filterValue.toLocaleLowerCase();
        return field.indexOf(filter) >= 0;
    });
  }
}
```

Если у вас нет имени поля или значения фильтра, то не выполняйте фильтрацию

Фильтрует массив объектов Product соответственно свойству, переданному в параметре filterByField. Возвращает значение true только для тех элементов массива, которые содержат символы, предоставленные как filterValue

2. Измените компонент `HomeComponent` так, чтобы он использовал `FilterPipe`. Компонент `HomeComponent` является предком для элементов `<auction-product-item>`, а также имеет переменную `products`, в которой хранится массив продуктов, предоставляемых сервисом `ProductService`. Канал `FilterPipe` будет отфильтровывать элементы этого массива.

   Вам нужно добавить элемент `<input>`, куда пользователь может вводить критерий для фильтрации. Компонент `HomeComponent` подпишется на наблюдаемый поток событий, исходящий из этого поля ввода, чтобы получать значение для фильтра.

Наконец, вам нужно использовать пользовательский канал; это включает в себя следующие действия:

- импорт `FilterPipe`;
- включение `FilterPipe` в раздел `declarations` аннотации `@NgModule`;
- применение канала в цикле `*ngFor` для каждого элемента массива.

Измените код файла `home.ts`, чтобы он выглядел так, как показано в листинге 5.9.

Листинг 5.9. Обновленный файл home.ts

```
import {Component} from '@angular/core';
import {FormControl} from '@angular/forms';
import {Product, ProductService} from '../../services/product-service';
import CarouselComponent from '../carousel/carousel';
import ProductItemComponent from '../product-item/product-item';
import {FilterPipe} from '../pipes/filter-pipe'
import 'rxjs/add/operator/debounceTime';

@Component({
  selector: 'auction-home-page',
  styleUrls: ['app/components/home/home.css'],
  template: `
    <div class="row carousel-holder">
      <div class="col-md-12">
        <auction-carousel></auction-carousel>
      </div>
    </div>
    <div class="row">
      <div class="col-md-12">
        <div class="form-group">
          <input placeholder="Filter products by title"
                 class="form-control" type="text"
                 [formControl]="titleFilter">
        </div>
      </div>
    </div>
    <div class="row">
      <div *ngFor="let product of products | filter:'title':filterCriteria"
class="col-sm-4 col-lg-4 col-md-4">
        <auction-product-item [product]="product"></auction-product-item>
      </div>
    </div>
  `
})
export default class HomeComponent {
  products: Product[] = [];
  titleFilter: FormControl = new FormControl();
  filterCriteria: string;

  constructor(private productService: ProductService) {
    this.products = this.productService.getProducts();
```

Импортирует канал

Добавляет элемент ввода, где пользователь может вводить текст

Применяет фильтр filter. Это имя пользовательского канала; title — свойство продукта, по которому нужно выполнить фильтрацию, а filterCriteria — имя свойства, хранящее текущий критерий фильтрации, введенный пользователем

```
    this.titleFilter.valueChanges
      .debounceTime(100)
      .subscribe(
        value => this.filterCriteria = value,
        error => console.error(error));
  }
}
```

Подписывается на поток событий ввода и указывает, что значение канала должно соответствовать текущему значению элемента <input>

3. Добавьте модуль `ReactiveFormsModule` в раздел `imports` директивы `@NgModule` файла `app.module.ts`. Он нужен для того, чтобы у вас был доступ к элементу `filter <input>`.
4. Запустите приложение, введя команду `npm start` в командной строке. Начните вводить какое-нибудь название продукта в поле `filter` и увидите, что браузер отрисовывает только те продукты, которые соответствуют критерию фильтра.

Для сравнения, мы запустили приложение-аукцион и приостановили его в отладчике после того, как ввели букву f в поле `filter`. На рис. 5.11 показано значение, полученное фильтром `FilterPipe` до того, как была выполнена фильтрация.

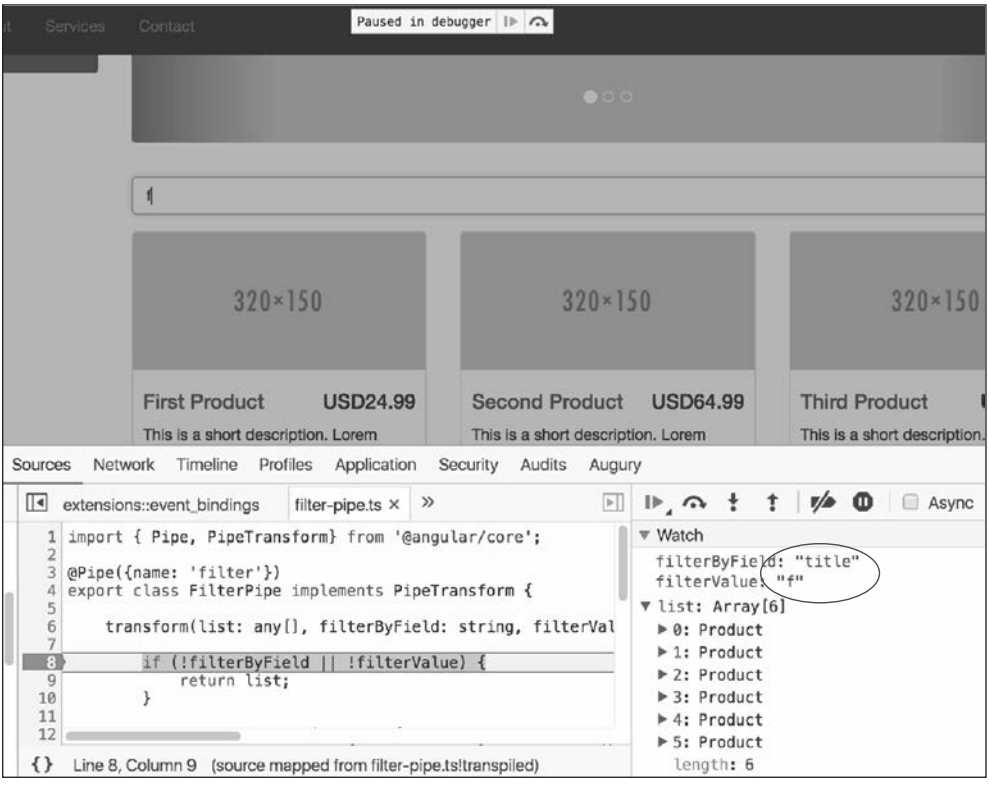


Рис. 5.11. Фильтр FilterPipe получает значение values

На панели Watch (Наблюдаемые значения) вы можете увидеть значения, полученные фильтром `FilterPipe` в качестве параметров. Стрелки показывают в действии деструктурирование TypeScript. После завершения фильтрации в этом окне показываются только те продукты, которые содержат в своих названиях букву f. В данном упражнении вы реализовали наблюдаемый поток событий и пользовательский фильтр.

5.5. Резюме

С точки зрения привязки данных обязанности между разработчиком и Angular разделяются. Первый отвечает за реализацию привязки между шаблоном компонента и поддерживающим кодом. Второй применяет свой механизм определения изменений для гарантии мгновенного обновления привязок с целью отразить текущее состояние приложения.

Наблюдаемые потоки являются фундаментальной концепцией стиля реактивного программирования, который используют разработчики, пишущие код на множестве языков программирования. RxJS 5, библиотека реактивных элементов JavaScript, интегрирована во фреймворк Angular.

Вот основные выводы этой главы.

- Привязка к свойству компонента позволяет отправить данные в одном направлении: от модели DOM к интерфейсу.
- Привязка к событиям помогает передать действия от интерфейса к компоненту.
- Двухсторонняя привязка обозначается нотацией `[()]`.
- Структурную директиву `ngIf` можно использовать для добавления и удаления узлов из модели DOM браузера.
- Применение наблюдаемых потоков данных упрощает асинхронное программирование. Можно подписаться на поток и отписаться от него, а также отменить ожидающие запросы данных.

6 Реализация коммуникации между компонентами

В этой главе:

- создание слабо связанных компонентов;
- передача данных компонентом-предком компоненту-потомку и наоборот;
- реализация шаблона проектирования;
- посредник для создания компонентов, которые можно использовать повторно;
- жизненный цикл компонента;
- разбираем определение изменений.

Мы установили, что любое Angular-приложение представляет собой дерево компонентов. При их разработке нужно убедиться в возможности их повторного использования и их самостоятельности, а также в том, что они имеют средство коммуникации с другими элементами. В этой главе мы рассмотрим, как компоненты могут передавать данные друг другу в слабо связанной манере.

Сначала мы покажем, как компонент-предок может передавать данные своим потомкам путем привязки к входным свойствам потомка. Далее вы увидите, как компонент-потомок может отправлять данные, генерируя события с помощью выходных свойств.

Мы продолжим тему, приведя пример использования шаблона проектирования «Посредник» с целью реализовать обмен данными между компонентами, не связанными отношениями «предок — потомок». Посредник — это, возможно, самый важный шаблон в любом фреймворке, в основе которого лежат элементы. Наконец, мы рассмотрим жизненный цикл компонента Angular и привязки, пригодные для написания необходимого приложению кода, перехватывающего важные события во время создания элемента, а также его жизненного цикла и разрушения.

6.1. Коммуникация между компонентами

На рис. 6.1 показано представление, состоящее из нескольких компонентов, которые пронумерованы и имеют разные формы (чтобы было проще на них ссылаться). Отдельные компоненты содержат другие элементы (назовем внешние компоненты контейнерами), другие же являются компонентами одного уровня. Чтобы абстрагироваться от конкретных фреймворков пользовательского интерфейса, мы избе-

гали использования таких элементов HTML, как поля ввода, выпадающие списки и кнопки, но вы можете перенести этот пример на представление вашего собственного приложения.

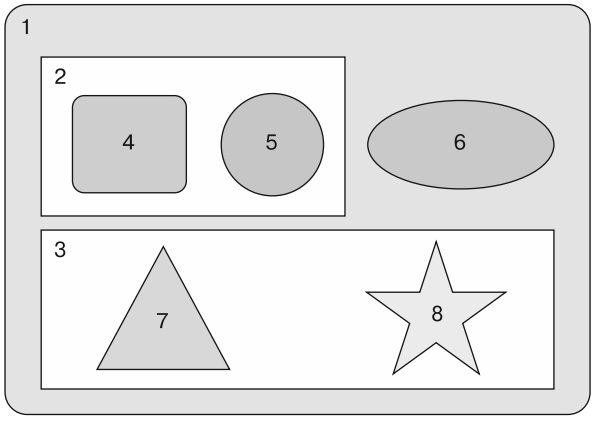


Рис. 6.1. Представление состоит из компонентов

Когда вы разрабатываете представление, которое содержит несколько компонентов, чем меньше эти компоненты знают друг о друге, тем лучше. Предположим, пользователь нажимает кнопку в компоненте 4, что также вызывает выполнение каких-то действий в компоненте 5. Возможно ли реализовать этот сценарий так, чтобы компонент 4 не знал о компоненте 5? Да, возможно.

Вы уже видели примеры слабо связанных элементов, когда мы рассматривали внедрение зависимостей. Теперь мы покажем другой способ достижения этой же цели: привязки и события.

6.1.1. Входные и выходные свойства

Представьте, что компонент Angular — это черный ящик, имеющий несколько выходов. Часть их отмечены как `@Input()`, а другие — как `@Output()`. Вы можете создать компонент, который будет иметь столько входных и выходных свойств, сколько захотите.

Если компонент Angular должен получать значения из внешнего мира, то можно связать производителей этих значений с соответствующими входными свойствами компонента. От кого получены данные значения? Элемент не должен это знать — ему достаточно информации о том, *что* с ними делать при получении.

Если компонент должен передавать значения во внешний мир, то может *испускать события* с помощью своих выходных свойств. Для кого он отправляет данные события? Компонент не должен это знать. Тот, кому это будет интересно, может слушать или подписываться на события, отправляемые элементами.

Реализуем описанные принципы. Сначала вы создадите компонент `OrderComponent`, который может получать запросы на заказы из внешнего мира.

Входные свойства

Входные свойства компонента декорируются аннотацией `@Input` и используются для получения данных от родительского компонента. Представьте, что хотите создать элемент интерфейса, позволяющий размещать заказы на покупку акций. Он будет знать, как связаться с фондовой биржей, но это неважно с точки зрения нашей дискуссии о входных свойствах. Вы хотите гарантировать, что компонент `OrderComponent` получает данные от других элементов с помощью свойств, отмеченных аннотацией `@Input`.

В листинге 6.1 показаны два компонента: `AppComponent` (предок) и `OrderComponent` (потомок). Второй имеет два свойства, `stockSymbol` и `quantity`, они отмечены аннотациями `@Input`. Компонент `AppComponent` позволяет пользователям ввести комбинацию символов, представляющую название акции, которая будет передана компоненту `OrderComponent` с помощью привязок.

Вы также будете передавать компоненту `OrderComponent` значение свойства `quantity`; но здесь вы не будете применять привязки, чтобы увидеть случай, когда предку нужно передать потомку значение, которое не будет изменяться. Вы не будете пользоваться механизмом привязок, не окружая атрибут `quantity` тегом `<order-processor>` с квадратными скобками.

Листинг 6.1. Содержимое файла input_property_binding.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule, Component, Input } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

@Component({
    selector: 'order-processor',
    template: `
    Buying {{quantity}} shares of {{stockSymbol}}
  `,
    styles:[`:host {background: cyan;}`]
})
class OrderComponent {

    @Input() stockSymbol: string;
    @Input() quantity: number;
}
@Component({
    selector: 'app',
    template: `
    <input type="text" placeholder="Enter stock (e.g. IBM)"
      ➥ (change)="onInputEvent($event)">
    <br/>
    <order-processor [stockSymbol]="stock"
            quantity="100"></order-processor>
  `
})
class AppComponent {
    stock: string;
    onInputEvent({target}):void{
```

Объявляет два входных свойства

Привязывает значение свойства stock компонента AppComponent к входному свойству компонента OrderComponent

Присваивает значение 100 входному свойству компонента OrderComponent. Здесь привязка не используется

```
        this.stock=target.value;
    }
}

@NgModule({
    imports: [ BrowserModule],
    declarations: [ AppComponent, OrderComponent],
    bootstrap: [ AppComponent ]
})
class AppModule { }

platformBrowserDynamic().bootstrapModule(AppModule);
```

Как только пользователь переместит фокус из входного свойства компонента AppComponent, будет отправлено событие change, и OrderComponent получит новую комбинацию символов для обработки

СОВЕТ

Поскольку вы не будете использовать привязку для атрибута quantity, значение 100 поступит в компонент OrderComponent как строка (все значения атрибутов HTML являются строками). Если хотите сохранить типы, то задействуйте привязки, например, так: [quantity]="100".

ПРИМЕЧАНИЕ

Если измените значение свойств stockSymbol или quantity в компоненте OrderComponent, то это не затронет значения свойств компонента-предка. Привязки свойств являются односторонними: от предка к потомку.

На рис. 6.2 показано окно браузера после того, как пользователь введет значение `IBM`. Компонент `OrderComponent` получил входные значения.

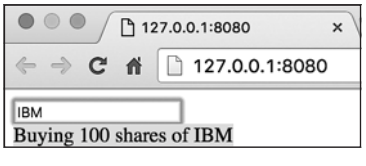


Рис. 6.2. Компонент OrderComponent получил значения

Следующий вопрос заключается в том, как компонент определяет момент, когда изменяется одно из его входных свойств? Самым простым способом будет изменить входное свойство так, чтобы оно стало сеттером. Вы используете `stockSymbol` в шаблоне компонента, поэтому вам также понадобится геттер. Поскольку ваш сеттер открытый, задайте переменной имя `_stockSymbol` и сделайте ее закрытой (листинг 6.2).

Листинг 6.2. Добавление сеттера и геттера

```
private _stockSymbol: string;
@Input()
set stockSymbol(value: string) {
    this._stockSymbol = value;
    if (this._stockSymbol != undefined) {
      console.log(`Sending a Buy order to NASDAQ:
      ➥ ${this.stockSymbol} ${this.quantity}`);
    }
}
get stockSymbol(): string {
    return this._stockSymbol;
}
```

Когда данное приложение запускается, все входные переменные инициализируются с помощью значений по умолчанию и механизм обнаружения изменений расценивает это как изменение привязанной переменной `stockSymbol`. Вызывается сеттер, и во избежание отправки `order` для `undefinedstockSymbol` вы проверяете его значение в сеттере.

ПРИМЕЧАНИЕ

В подразделе 6.2.1 мы опишем, как перехватывать изменения входных свойств, не используя сеттеры.

В главе 3 мы показали, как передавать параметры в компонент, используя `ActivatedRoute`. В этом сценарии параметры передаются с помощью конструктора. Привязка к параметрам `@Input()` — решение для передачи данных от предка к потомку, оно работает только в том случае, если компоненты располагаются внутри одного маршрута.

Выходные свойства и пользовательские события

Компоненты Angular могут отправлять пользовательские события, используя объект `EventEmitter`. Эти события можно обрабатывать либо в компоненте, либо с помощью его предков. Данный объект представляет собой подкласс `Subject` (реализован в библиотеке RxJS), который может быть как наблюдаемым потоком, так и наблюдателем. Другими словами, `EventEmitter` может отправлять пользовательские события с помощью метода `emit()`, а также работать с наблюдаемыми потоками, задействуя метод `subscribe()`. Поскольку этот раздел посвящен отправке данных элемента во внешний мир, мы сконцентрируемся на теме отправки пользовательских событий.

Предположим, что вам нужно написать компонент интерфейса, который связан с фондовой биржей и отображает изменяющиеся цены на акции. Он может быть использован в приложении для работы с финансами какой-нибудь брокерской фирмы. Помимо отображения цен, компонент также должен отправлять события, содержащие последние цены, во внешний мир, чтобы другие элементы могли применить к изменяющимся ценам бизнес-логику.

Создадим компонент `PriceQuoterComponent`, который реализует подобную функциональность. Для этого примера вам нужно не связываться с серверами, а только эмулировать изменяющиеся цены, используя генератор случайных чисел. Отображать изменяющиеся цены в компоненте `PriceQuoterComponent` можно довольно прямолинейно: вы свяжете свойства `stockSymbol` и `lastPrice` с шаблоном компонента.

Внешний мир вы оповестите путем отправки пользовательских событий с помощью свойства элемента с аннотацией `@Output`. Вы будете отправлять событие, как только изменится цена. Оно будет нести полезную нагрузку: объект, содержащий комбинацию символов для акции и ее последнюю цену. Эта функциональность реализована в следующем сценарии (листинг 6.3).

Листинг 6.3. Содержимое файла output-property-binding.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule, Component, Output, EventEmitter }
➥      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

interface IPriceQuote {
    stockSymbol: string;
    lastPrice: number;
}

@Component({
    selector: 'price-quoter',
    template: `<strong>Inside PriceQuoterComponent: {{stockSymbol}}
 {{price | currency:'USD':true:'1.2-2'}}</strong>`,
    styles:[`:host {background: pink;}`]
})

class PriceQuoterComponent {
    @Output() lastPrice: EventEmitter
    ➥ <IPriceQuote> = new EventEmitter();

    stockSymbol: string = "IBM";
    price:number;

    constructor() {
      setInterval(() => {
          let priceQuote: IPriceQuote = {
              stockSymbol: this.stockSymbol,
              lastPrice: 100*Math.random()
          };

          this.price = priceQuote.lastPrice;

          this.lastPrice.emit(priceQuote)
      }, 1000);
    }
}

@Component({
   selector: 'app',
   template: `
   <price-quoter (lastPrice)="priceQuoteHandler($event)">
   ➥</price-quoter><br>
   AppComponent received: {{stockSymbol}} {{price |
   ➥ currency:'USD':true:'1.2-2'}}`
})

@NgModule({
    imports: [ BrowserModule],
    declarations: [ AppComponent, PriceQuoterComponent],
```

Объявляет интерфейс TypeScript, чтобы представить предложения по ценам. Это поможет вашей IDE выполнить проверку на наличие ошибок и опережающего ввода

Выводит сочетание символов и цену в интерфейсе компонента. Для форматирования валют вы используете CurrencyPipe

Выходное свойство lastPrice представлено событием EventEmitter, которое генерирует события lastPrice для предков этого компонента

В этом примере в качестве комбинации символов для акции используется жестко закодированное значение IBM

Эмулирует изменяющиеся цены путем вызова функции, которая генерирует случайное число каждую секунду и заполняет объект priceQuote

Отправляет каждую новую цену всем, кому это может быть интересно, с помощью выходного свойства путем генерации события, несущего в качестве полезной нагрузки объект priceQuot

Тег представляет собой компонент-потомок PriceQuoterComponent, который обновляет предложения по ценам в этом шаблоне. На уровне приложения обработчик событий для события last-price вызывается для того, чтобы отобразить предложения по цене, поступающей вместе с объектом события

Отображает предложения по цене и в шаблоне приложения

```
    bootstrap: [ AppComponent ]
})
class AppModule { }

platformBrowserDynamic().bootstrapModule(AppModule);
```

Обработчик событий получает объект типа `IPriceQuote`, и вы извлекаете из него значения свойств `stockSymbol` и `lastPrice`. Если запустите этот пример, то увидите, как цены обновляются каждую секунду в компонентах `PriceQuoterComponent` (затененный фон) и `AppComponent` (белый фон), как показано на рис. 6.3.

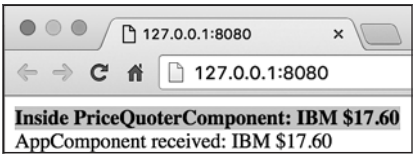


Рис. 6.3. Запуск примера работы с выходными свойствами

СОВЕТ

По умолчанию имя пользовательского события совпадает с именем выходного свойства, в нашем случае это lastPrice. Если хотите генерировать событие с другим именем, то укажите его в качестве аргумента для аннотации @Output. Например, чтобы сгенерировать событие с именем last-price, объявите выходное свойство как @Output('last-price') lastPrice;.

В листинге 6.3 вы создаете компонент Angular `PriceQuoterComponent`, который содержит пользовательский интерфейс. Но для бизнеса может понадобиться функциональность получения `price-quote` без интерфейса, чтобы использовать ее как в приложениях для трейдеров, так и в крупных панелях инструментов. Вы можете реализовать ту же функциональность в качестве внедряемого сервиса, как сделали для сервиса `ProductService` в проекте онлайн-аукциона.

Поднятие событий

На момент написания данной книги Angular не предлагает синтаксис, поддерживающий поднятие событий. Для компонента `PriceQuoterComponent` это значит следующее: если вы хотите отслеживать событие `last-price` не в самом компоненте, а в его предке, то событие туда не поднимется. В следующем фрагменте кода событие `last-price` не достигнет элемента `<div>`, поскольку он является предком элемента `<price-quoter>`:

```
<div (last-price)="priceQuoteHandler($event)">
  <price-quoter ></price-quoter>
</div>
```

Если для вашего приложения важна поддержка поднятия событий, то не используйте переменную `EventEmitter`; вместо этого задействуйте нативные события модели DOM. В следующем примере показана еще одна версия компонента `PriceQuoterComponent`, в которой поднятие событий обрабатывается без `EventEmitter`:

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule, Component, ElementRef }        from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
```

```
interface IPriceQuote {
  stockSymbol: string,
  lastPrice: number
}

@Component({
  selector: 'price-quoter',
  template: `PriceQuoter: {{stockSymbol}} \${{price}}`,
  styles:[`:host {background: pink;}`]
})
class PriceQuoterComponent {
  stockSymbol: string = "IBM";
  price:number;
  constructor(element: ElementRef) {
    setInterval(() => {Inter-component communication
      let priceQuote: IPriceQuote = {
        stockSymbol: this.stockSymbol,
        lastPrice: 100*Math.random()
      };
      this.price = priceQuote.lastPrice;
      element.nativeElement
          .dispatchEvent(new CustomEvent('last-price', {
            detail: priceQuote,
            bubbles: true
          }));
    }, 1000);
  }
}

@Component({
  selector: 'app',
  template: `
    <div (last-price)="priceQuoteHandler($event)">
      <price-quoter></price-quoter>
    </div>
    <br>
    AppComponent received: {{stockSymbol}} \${{price}}
  `
})
class AppComponent {
  stockSymbol: string;
  price:number;
  priceQuoteHandler(event: CustomEvent) {
    this.stockSymbol = event.detail.stockSymbol;
    this.price = event.detail.lastPrice;
  }
}

@NgModule({
  imports:      [ BrowserModule],
  declarations: [ AppComponent, PriceQuoterComponent],
  bootstrap:    [ AppComponent ]
```

```
})
class AppModule { }
platformBrowserDynamic().bootstrapModule(AppModule);
```

В предыдущем приложении Angular внедрил ссылку на элемент модели DOM, который представляет собой элемент `<price-quoter>`, с помощью `ElementRef`, а затем путем вызова метода `element.nativeElement.dispatchEvent()` было отправлено пользовательское событие. Поднятие здесь сработает, но помните, что этот код может не функционировать в ряде браузеров и для всех отрисовщиков, не поддерживающих HTML.

6.1.2. Шаблон «Посредник»

Когда вы разрабатываете пользовательский интерфейс на основе компонентов, каждый из них должен работать самостоятельно и не полагаться на существование других UI-элементов. Такие слабо связанные компоненты можно реализовать с помощью шаблона проектирования «Посредник», который, согласно «Википедии», «обеспечивает взаимодействие множества объектов» (https://en.wikipedia.org/wiki/Mediator_pattern). Мы объясним, что это значит, с помощью аналогии с детским конструктором.

Представьте, что ребенок играет с конструктором, то есть *компонентами*, которые не знают друг о друге. Сегодня этот ребенок *(посредник)* может использовать кирпичики для постройки дома, а завтра из тех же элементов он сделает лодку.

ПРИМЕЧАНИЕ

Задача посредника — гарантировать, что компоненты будут подходить друг другу в зависимости от задачи и при этом останутся слабо связанными.

Снова взглянем на рис. 6.1. Каждый компонент, за исключением 1, имеет предка (контейнер), который играет роль посредника. Посредник верхнего уровня — это контейнер 1, он отвечает за то, что компоненты 2, 3 и 6 могут общаться друг с другом, если нужно. С другой стороны, компонент 2 является посредником для компонентов 4 и 5. Компонент 3 является посредником для компонентов 7 и 8.

Посредник должен получать данные из одного компонента и передавать их другому. Вернемся к примерам, связанным с наблюдением за ценами на акции.

Представьте трейдера, наблюдающего за ценами на некоторые акции. В какой-то момент он нажимает кнопку Buy (Купить), расположенную рядом с одной из акций, чтобы сделать заказ на покупку на фондовой бирже. Вы легко можете добавить эту кнопку в компонент `PriceQuoterComponent` из предыдущего раздела, но данный компонент не знает, как размещать заказы на покупку акций. Он оповестит посредника (`AppComponent`), что трейдер хочет купить определенные акции именно сейчас.

Посредник должен знать, какой элемент может размещать заказы и как передать ему название акций и требуемое количество. На рис. 6.4 показано, как компонент `AppComponent` может быть посредником в коммуникации между `PriceQuoterComponent` и `OrderComponent`.

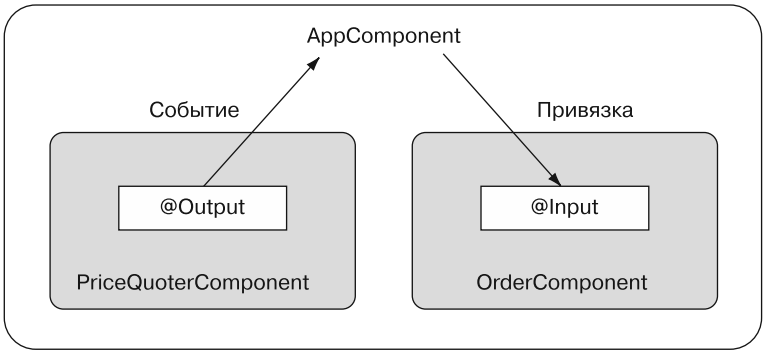


Рис. 6.4. Посредник при коммуникации

ПРИМЕЧАНИЕ

Отправка событий работает в широковещательном режиме. Компонент PriceQuoterComponent отправляет события с помощью свойства @Output, не зная, кто будет получать события. Компонент OrderComponent ожидает изменения значения в свойстве @Input — это сигнал о размещении заказа.

Чтобы показать шаблон «Посредник» в действии, напишем небольшое приложение, состоящее из двух компонентов, показанных на рис. 6.4. Вы можете найти это приложение в каталоге `mediator`, в котором содержатся следующие файлы:

- `stock.ts` — интерфейс, в котором определяется объект `value`, представляющий собой акцию;
- `price-quoter.ts` — компонент `PriceQuoterComponent`;
- `order.ts` — компонент `OrderComponent`;
- `mediator.ts` — компоненты `PriceQuoterComponent` и `OrderComponent`.

Вы будете использовать интерфейс `Stock` в двух случаях:

- чтобы представить полезную нагрузку события, отправленного компонентом `PriceQuoteComponent`;
- для представления данных, переданных компоненту `OrderComponent` с помощью привязки.

Далее рассмотрим содержимое файла `stock.ts` (листинг 6.4).

Листинг 6.4. Содержимое файла stock.ts

```
export interface Stock {
  stockSymbol: string;
  bidPrice: number;
}
```

Предположим, вы хотите использовать SystemJS для динамической компиляции кода TypeScript. По умолчанию SystemJS превратит содержимое файла `stock.ts`

в пустой модуль stock.js, и вы получите ошибку, когда загрузчик SystemJS попробует импортировать его. Вам нужно дать SystemJS знать о том, что Stock следует рассматривать как модуль. Это можно сделать, сконфигурировав SystemJS с помощью метааннотации, как показано в следующем фрагменте кода, взятом из файла systemjs.config.js:

```
packages: {...},
meta: {
  'app/mediator/stock.ts': {
    format: 'es6'
  }
}
```

Компонент PriceQuoteComponent, показанный далее (листинг 6.5), имеет кнопку Buy (Купить), а также выходное свойство buy. Оно отправляет событие buy только в том случае, когда пользователь нажимает эту кнопку.

Листинг 6.5. Содержимое файла price-quoter.ts

```
import {Component, Output, Directive, EventEmitter} from '@angular/core';
import {Stock} from './stock';

@Component({
    selector: 'price-quoter',
    template: `<strong><input type="button" value="Buy"
    ➥ (click)="buyStocks($event)">
        {{stockSymbol}} \${{lastPrice | currency:'USD':true:'1.2-2'}}
        ➥ </strong>
            `,
    styles:[`:host {background: pink; padding: 5px 15px 15px 15px;}`]
})
export class PriceQuoterComponent {
    @Output() buy: EventEmitter <Stock> = new EventEmitter();
    stockSymbol: string = "IBM";
    lastPrice:number;
    constructor() {
        setInterval(() => {
            this.lastPrice = 100*Math.random();
        }, 2000);
    }
    buyStocks(): void{
        let stockToBuy: Stock = {
            stockSymbol: this.stockSymbol,
            bidPrice: this.lastPrice
        };
        this.buy.emit(stockToBuy);
    }
}
```

Когда посредник (AppComponent) получает событие buy из элемента <price-quoter>, он извлекает из него полезную нагрузку и присваивает ее переменной

stock, которая связана с входным параметром <order-processor>. Код показан далее (листинг 6.6).

Листинг 6.6. Содержимое файла mediator.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule, Component} from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import {OrderComponent} from './order';
import {PriceQuoterComponent} from './price-quoter';
import {Stock} from './stock';

@Component({
    selector: 'app',
    template: `
    <price-quoter (buy)="priceQuoteHandler($event)"></price-quoter><br>
    <br/>
    <order-processor [stock]="stock"></order-processor>
  `
})
class AppComponent {
    stock: Stock;
    priceQuoteHandler(event:Stock) {
        this.stock = event;
    }
}

@NgModule({
    imports:      [ BrowserModule],
    declarations: [ AppComponent, OrderComponent,
                    PriceQuoterComponent],
    bootstrap:    [ AppComponent ]
})
class AppModule { }
platformBrowserDynamic().bootstrapModule(AppModule);
```

Когда значение входного свойства buy в компоненте OrderComponent изменяется, его сеттер отображает сообщение Placed order..., показывая stockSymbol и bidPrice (листинг 6.7).

Листинг 6.7. Содержимое файла order.ts

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

@Component({
    selector: 'order-processor',
    template: `{{message}}`,
    styles:[`:host {background: cyan;}`]
})
export class OrderComponent {
    message:string = "Waiting for the orders...";
    private _stock: Stock;
```

```
    @Input() set stock(value: Stock ){
        if (value && value.bidPrice != undefined) {
            this.message = `Placed order to buy 100 shares of
            ➥ ${value.stockSymbol} at \$${value.bidPrice.toFixed(2)}`;
        }
    }
    get stock(): Stock{
        return this._stock;
    }
}
```

Снимок экрана, показанный на рис. 6.5, был сделан после того, как пользователь нажал кнопку Buy (Купить) в момент, когда цена на акции IBM составляла $12,17. Компонент `PriceQuoteComponent` отрисован в верхней части экрана, а `OrderComponent` — в нижней. Эти элементы самостоятельны и слабо связаны.

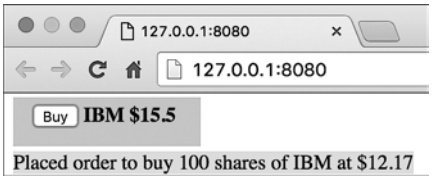


Рис. 6.5. Запуск примера с посредником

СОВЕТ

Не начинайте реализовывать компоненты интерфейса вашего приложения до тех пор, пока не определите посредников, пользовательские повторно используемые компоненты и средства коммуникации между ними.

Шаблон проектирования «Посредник» также хорошо подходит для онлайн-аукциона. Представьте, что идут последние минуты войны ставок за какой-нибудь популярный товар. Пользователи следят за постоянно обновляющимися ставками и нажимают кнопку, чтобы повысить свою ставку.

Альтернативная реализация шаблона «Посредник»

В этом разделе вы увидели, как компоненты одного уровня используют своих предков в качестве посредников. Если компоненты не имеют общего предка или не отображаются одновременно (маршрутизатор может не отображать требуемый элемент в данный момент), то можно применять в качестве посредника внедряемый сервис. При создании компонента в него внедряется сервис-посредник, и компонент может подписаться на события, отправляемые сервисом (в противоположность использованию параметров `@Input()`, как мы делали в компоненте `OrderComponent`).

Если вы хотите увидеть представленный подход в действии, то прочтите раздел «Предоставление результатов поиска компоненту HomeComponent» упражнения в главе 8. Взгляните на код сервиса `ProductService`, играющий роль посредника. В этом сервисе определяется `searchEvent`: переменная `EventEmitter` используется компонентом `SearchComponent` для отправки данных, введенных пользователем. Компонент `HomeComponent` подписывается на переменную `searchEvent` для того, чтобы получать текст, который пользователь вводит в форме поиска.

6.1.3. Изменяем шаблоны во время работы программы с помощью ngContent

В некоторых случаях нужно динамически изменить содержимое компонента шаблона во время работы программы. В AngularJS эта функциональность называлась виртуальным включением, но теперь она известна как «проекция». В Angular можно спроецировать фрагмент шаблона компонента-предка на потомка с помощью директивы ngContent. Синтаксис довольно прост, вам нужно сделать всего два шага.

1. В шаблоне компонента-потомка включите теги `<ng-content></ng-content>` (точка внедрения).
2. В компоненте-предке включите фрагмент HTML, который вы хотите спроецировать на точку внедрения в потомке, между тегами, представляющими компонент-потомок (например, `<my-child>`):

```
template: `
    ...
    <my-child>
      <div>Passing this div to the child</div>
    </my-child>
    ...
`
```

В этом примере компонент-предок не будет отрисовывать содержимое, которое находится внутри тегов `<my-child>` и `</my-child>`. В листинге 6.8 показано применение данного приема.

Листинг 6.8. Содержимое файла basic-ng-content.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule, Component, ViewEncapsulation} from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

@Component({
  selector: 'child',
  styles: ['.wrapper {background: lightgreen;}'],
  template: `
    <div class="wrapper">
     <h2>Child</h2>
      <div>This div is defined in the child's template</div>
      <ng-content></ng-content>
    </div>
  `,
  encapsulation: ViewEncapsulation.Native
})
class ChildComponent {}

@Component({
  selector: 'app',
  styles: ['.wrapper {background: cyan;}'],
  template: `
    <div class="wrapper">
     <h2>Parent</h2>
```

Содержимое, которое поступает от предка, отображается здесь

По умолчанию Angular использует режим ViewEncapsulation.Emulated (см. врезку «Поддержка Shadow DOM в Angular» в главе 3). Вы начнете с режима Native, а затем запустите данную программу снова в режимах Emulated и None

```
      <div>This div is defined in the Parent's template</div>
      <child>
        <div>Parent projects this div onto the child </div>
      </child>
    </div>
  `,
  encapsulation: ViewEncapsulation.Native
})
class AppComponent {}

@NgModule({
  imports: [ BrowserModule],
  declarations: [ AppComponent, ChildComponent],
  bootstrap: [ AppComponent ]
})
class AppModule { }

platformBrowserDynamic().bootstrapModule(AppModule);
```

Это содержимое не отрисовывается в компоненте AppComponent, оно передается компоненту ChildComponent

Мы также используем данный пример для демонстрации того, как работают Shadow DOM и `ViewEncapsulation`. Вы заметили, что и предок и потомок используют стиль `.wrapper` для внешнего элемента `<div>`? На обычной странице HTML это означало бы следующее: предок и потомок были бы отрисованы с применением одного стиля. Мы покажем такие способы инкапсуляции стилей компонентов-потомков, которые не будут конфликтовать со стилями предков, имеющими те же имена.

На рис. 6.6 показано запущенное приложение в режиме `ViewEncapsulation.Native` при открытой панели Developer Tools (Инструменты разработчика). Компонент `ChildComponent` получил содержимое HTML из компонента `AppComponent` и создал узлы Shadow DOM для предка и потомка (взгляните на элемент `#shadow-root`, расположенный справа). Обратите внимание на то, что стиль `.wrapper` из родительского элемента `<div>` (голубой фон) не был применен к элементу `<div>` потомка, также использующего стиль `.wrapper`, — он отрисовывается со светло-зеленым фоном. `#shadow-root` потомка выполняет роль стены, защищающей стили потомка, не давая наследовать стили предка.

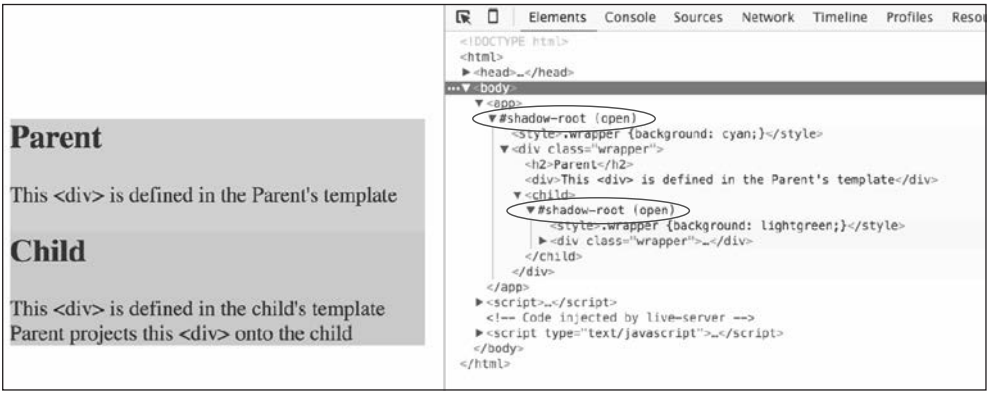


Рис. 6.6. Запуск файла basic-ng-content.ts в режиме ViewEncapsulation.Native

Снимок экрана, показанный на рис. 6.7, был сделан после изменения режима инкапсуляции на `ViewEncapsulation.Emulated`. Структура DOM отличается от показанной ранее, и узлов `#shadow-root` больше нет. Angular сгенерировал дополнительные атрибуты для элементов предка и потомка, чтобы реализовать инкапсуляцию, но интерфейс отрисовывается точно так же.

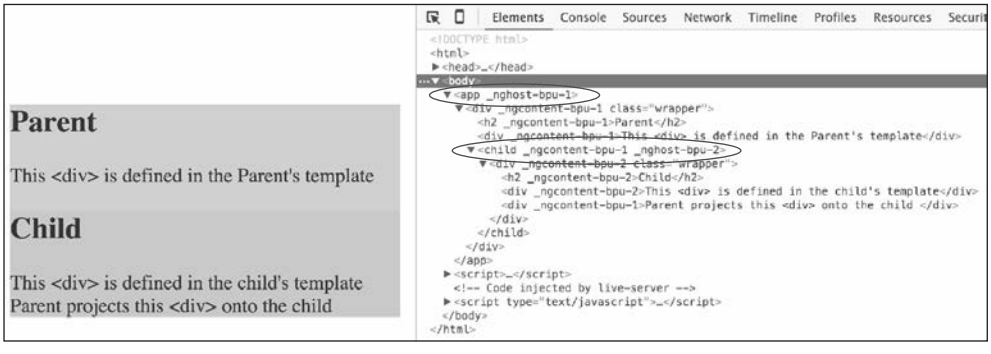


Рис. 6.7. Запуск файла basic-ng-content.ts в режиме ViewEncapsulation.Emulated

На рис. 6.8 показан тот же пример, запущенный в режиме `ViewEncapsulation.None`. В этом случае все элементы предка и потомка объединяются в основное дерево DOM, а стили не инкапсулируются — для всего окна используется светло-зеленый фон предка.

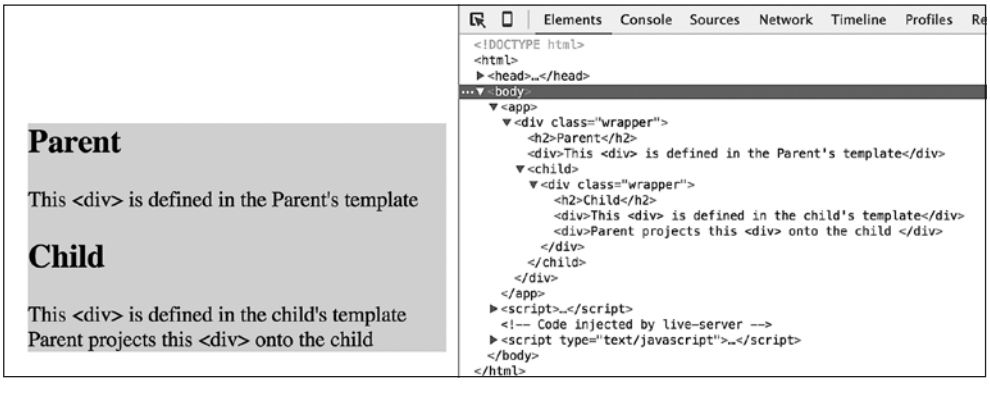


Рис. 6.8. Запуск примера basic-ng-content.ts в режиме ViewEncapsulation.None

Проецирование на несколько областей. Компонент может иметь в своем шаблоне больше одного тега `<ng-content>`. Рассмотрим пример, когда шаблон потомка разбит на три области: верхний и нижний колонтитулы, а также зону содержимого. Разметка HTML для колонтитулов может быть спроецирована предком, а область содержимого может быть определена в потомке. Чтобы это реализовать, нужно включить в элемент-потомок две отдельные пары тегов `<ng-content></ng-content>`, заполняемые предком (колонтитулы).

Для гарантии того, что содержимое колонтитулов будет отрисовано в соответствующих областях `<ng-content>`, будет использоваться атрибут `select`, который может быть любым корректным селектором (класс CSS, имя тега и т. д.). Шаблон потомка может выглядеть так:

```
<ng-content select=".header"></ng-content>
<div>This content is defined in child</div>
<ng-content select=".footer"></ng-content>
```

Содержимое, поступающее от предка, будет соотнесено с помощью селектора и отрисовано в соответствующей области. Рассмотрим код, в котором полностью реализована эта функциональность (листинг 6.9).

Листинг 6.9. Содержимое файла ng-content-selector.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule, Component}      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

@Component({
  selector: 'child',
  styles: ['.child {background: lightgreen;}'],
  template: `
    <div class="child">
    <h2>Child</h2>
      <ng-content select=".header" ></ng-content>
      <div>This content is defined in child</div>
      <ng-content select=".footer"></ng-content>
    </div>
  `
})
class ChildComponent {}

@Component({
  selector: 'app',
  styles: ['.app {background: cyan;}'],
  template: `
    <div class="app">
    <h2>Parent</h2>
      <div>This div is defined in the Parent's template</div>
      <child>
        <div class="header" >Child got this header from parent {{todaysDate}}
        ➥ </div>
        <div class="footer">Child got this footer from parent</div>
      </child>
    </div>
  `
})
class AppComponent {
  todaysDate: string = new Date().toLocaleDateString();
}

@NgModule({
  imports:      [ BrowserModule],
```

```
  declarations: [ AppComponent, ChildComponent],
  bootstrap:    [ AppComponent ]
})
class AppModule { }

platformBrowserDynamic().bootstrapModule(AppModule);
```

Обратите внимание: вы используете привязку свойств в компоненте `AppComponent`, чтобы включить в верхний колонтитул сегодняшнюю дату. Спроецированный код HTML может содержать привязки только для тех свойств, которые видны в области видимости предка, поэтому вы не можете применять свойства потомка в родительском выражении привязки.

В результате запуска данного примера будет отрисована страница, показанная на рис. 6.9. Директива `ngContent`, имеющая атрибут `select`, позволяет создать универсальный компонент, чье представление разбито на несколько областей, получающих разметку извне.

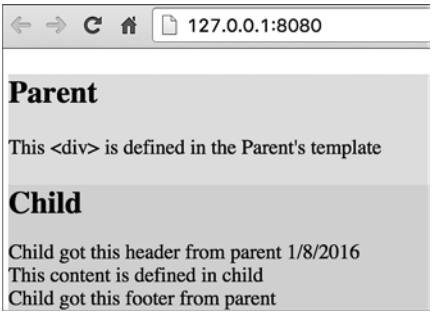


Рис. 6.9. Запуск файла ng-content-select.ts

Прямая привязка к innerHTML

Вы можете связать свойство компонента, имеющее содержимое HTML, непосредственно с шаблоном, как показано в примере ниже:

```
<p [innerHTML]="myComponentProperty"></p>
```

Но вместо этого предпочтительнее использовать директиву `ngContent` по следующим причинам:

- `innerHTML` — API, характерный для некоторых браузеров, а указанная директива не зависит от платформы;
- с помощью данной директивы можно определить несколько мест, куда будут добавлены фрагменты HTML;
- директива `ngContent` позволяет связать свойства компонента-предка и спроецированный HTML.

6.2. Жизненный цикл компонента

За время жизненного цикла с компонентом Angular случаются разные события. После создания элемента механизм определения изменений (он объясняется в следующем разделе) начинает за ним наблюдать. Компонент инициализируется, добавляется в модель DOM и отрисовывается, чтобы пользователь мог увидеть его. Затем состояние элемента (значения его свойств) может измениться, что вызовет перерисовку интерфейса; и наконец, компонент уничтожается.

На рис. 6.10 показаны привязки жизненного цикла (функции обратного вызова), где вы добавляете пользовательский код, если нужно. Функции обратного вызова, показанные на светло-сером фоне, будут вызваны всего один раз, а функции на темно-сером фоне — несколько раз.

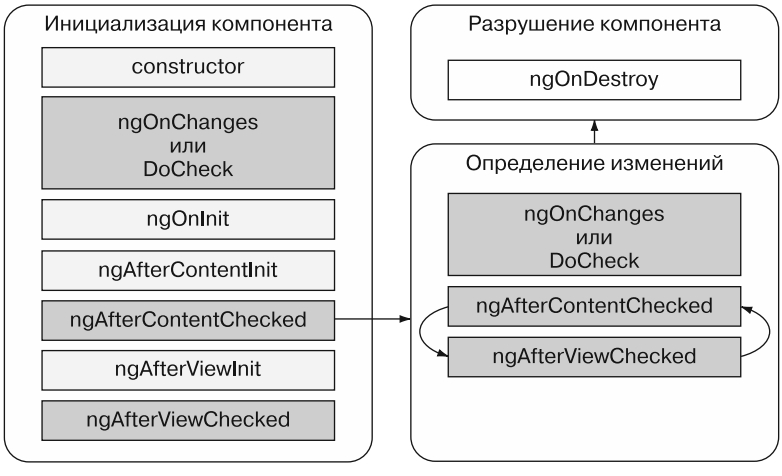


Рис. 6.10. Жизненный цикл компонента

Пользователь видит компонент после завершения фазы инициализации. Затем механизм определения изменения гарантирует, что свойства элемента будут синхронизированы с его интерфейсом. Если компонент удален из дерева DOM в результате навигации или работы структурной директивы (например, `ngIf`), то Angular инициирует фазу разрушения.

При создании экземпляра компонента первым вызывается конструктор, но во время его работы свойства еще не инициализированы. После завершения работы конструктора Angular вызовет следующие функции обратного вызова, *если вы их реализовали*.

- `ngOnChanges()` — вызывается, когда компонент-предок изменяет (или инициализирует) значения, связанные с входными свойствами потомка. При отсутствии у компонента таких свойств данная функция не вызывается. Если вы хотите реализовать собственный алгоритм определения изменений, то его нужно разместить в функции `DoCheck()`. Но реализация этого алгоритма может оказаться дорогой, поскольку указанная функция вызывается после каждого цикла определения изменений.
- `ngOnInit()` — вызывается после первого вызова функции `ngOnChanges()`, если таковые случаются. Несмотря на то что вы можете реализовать некоторые переменные компонента в конструкторе, свойства компонента еще не будут готовы. Однако к моменту вызова `ngOnInit()` они уже будут инициализированы.
- `ngAfterContentInit()` — вызывается, когда инициализируется состояние компонента-потомка, если вы использовали директиву `ngContent`, чтобы передать ей какой-то код HTML.

- `ngAfterContentChecked()` — вызывается для компонента-потомка, который применил директиву `ngContent` после того, как получил содержимое от предка (или во время фазы определения изменений), если в директиве `ngContent` были использованы привязки.
- `ngAfterViewInit()` — вызывается, когда завершается привязка для шаблона компонента. Сначала инициализируется родительский элемент, а затем, если у него есть потомки, эта функция вызывается после того, как все они будут готовы.
- `ngAfterViewChecked()` — вызывается, когда механизм определения изменений проверяет, имеются ли какие-то изменения в привязках шаблона компонента. Данная функция обратного вызова может быть вызвана больше одного раза из-за изменений в этом или других элементах.

Слово `Content` в имени метода обратного вызова жизненного цикла означает, что метод применяется, если содержимое проецируется с помощью директивы `<ng-content>`. Слово `View` в имени метода обратного вызова свидетельствует о его применении к шаблону компонента. Слово `Checked` означает, что изменения элемента применены и компонент синхронизируется с моделью DOM.

Для некоторых приложений может понадобиться вызывать определенную бизнес-логику, когда изменяется значение свойства. Например, финансовым приложениям может понадобиться записывать в журнал каждый шаг трейдера. Поэтому, если последний размещает заказ на покупку акций на сумму $101, а затем меняет сумму на $100, то данное изменение нужно записать в файл журнала. Приведенный пример хорошо показывает возможность использования журналирования в методе `DoCheck()`.

Когда не нужно писать код в конструкторах

В приложении онлайн-аукциона вы внедряете сервис `ProductService` в конструктор компонента `HomeComponent` и вызываете там метод `getProducts()`. Если методу `getProducts()` нужно использовать значения свойств элемента, то переместите вызов этого метода в метод `ngOnInit()` для гарантии того, что все свойства были инициализированы к моменту вызова `getProducts()`. Переместить код из конструктора в данный метод можно и по другой причине: это позволит оставить код конструктора легким, не размещая в нем никаких долгоиграющих синхронных функций.

На этапе разрушения объекта приложение может очищать системные ресурсы. Предположим, компонент подписан на сервис уровня приложения, который отслеживает состояние программы (например, магазин приложений, предлагаемый библиотекой Redux). Когда Angular уничтожает данный компонент, он должен отписаться от этого сервиса с помощью функции обратного вызова `ngOnDestroy()`.

ПРИМЕЧАНИЕ

Каждая функция обратного вызова жизненного цикла объявляется в интерфейсе с именем, соответствующим имени функции обратного вызова, но не имеет префикса ng. Например, если вы планируете реализовать какую-то функциональность в функции обратного вызова ngOnChanges(), то добавьте в объявление вашего класса конструкцию implements OnChanges.

Для получения более подробной информации о жизненном цикле компонента прочтите документацию Angular о привязках жизненного цикла на https://angular.io/guide/lifecycle-hooks. В следующем подразделе вы увидите пример, в котором применяется одна из привязок жизненного цикла.

6.2.1. Использование ngOnChanges

Проиллюстрируем привязки жизненного цикла компонента с помощью функции `ngOnChanges()`. В этом примере вы увидите элементы-предки и элементы-потомки, у последних будет два входных свойства: `greeting` и `user`. Первое свойство — строка, второе — объект с одним свойством, `name`. Чтобы понять, почему вызывается (или не вызывается) функция обратного вызова `ngOnChanges()`, нужно познакомиться с концепцией изменяемых и неизменяемых объектов.

Изменяемое против неизменяемого

Строки в JavaScript неизменяемы. Это значит, что созданную в памяти строку нельзя изменить. Рассмотрим следующий фрагмент кода:

```
var greeting = "Hello";
greeting = "Hello Mary";
```

В первой строке в определенной точке в памяти (а именно — по адресу `@287651`) создается значение `Hello`. Во второй строке значение по этому адресу не изменяется, поскольку создается новая строка `Hello Mary` в другом месте, а именно — по адресу `@286777`. Теперь в памяти содержится две строки, каждую из которых нельзя изменить.

Что происходит с переменной `greeting`? Ее значение изменяется, поскольку изначально она указывала на одну точку в памяти, а затем стала указывать на другую.

Объекты в JavaScript являются изменяемыми — это значит, что после создания объект остается в определенной точке в памяти, даже если изменяются значения их свойств. Рассмотрим следующий код:

```
var user = {name: "John"};
user.name = "Mary";
```

После выполнения первой строки кода создается объект, и переменная `user` указывает на определенную точку в памяти, `@277500`. Строка `John` создается в другой точке памяти, `@287600`, и переменная `user.name` сохраняет ссылку на этот адрес.

После выполнения второй строки кода создается новая строка `Mary` в третьей точке памяти, `@287700`, и переменная `user.name` сохраняет ссылку на этот новый адрес. Но переменная `user` все еще хранит адрес `@277500`. Другими словами, вы изменили содержимое объекта по адресу `@277500`.

Добавим в компонент-потомок привязку `ngOnChanges()`, чтобы показать, как она перехватывает изменения входных свойств. Это приложение имеет элементы-предки и элементы-потомки. Потомок имеет два входных свойства (`greetings` и `user`) и одно обычное (`message`). Пользователь может изменить значения вход-

ных свойств потомка. Покажем, какие значения свойств будут переданы методу `ngOnChanges()` в случае его вызова (листинг 6.10).

Листинг 6.10. Содержимое файла ng-onchanges-with-param.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule, Component, Input, OnChanges, SimpleChange,
➥ enableProdMode } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import {FormsModule} from '@angular/forms';

interface IChanges {[key: string]: SimpleChange};

@Component({
  selector: 'child',
  styles: ['.child{background:lightgreen}'],
  template: `
     <div class="child">
      <h2>Child</h2>
      <div>Greeting: {{greeting}}</div>
      <div>User name: {{user.name}}</div>
      <div>Message: <input [(ngModel)]="message"></div>
    </div>
  `
})
class ChildComponent implements OnChanges {
  @Input() greeting: string;
  @Input() user: {name: string};

  message: string = 'Initial message';

  ngOnChanges(changes: IChanges) {
    console.log(JSON.stringify(changes, null, 2));
  }
}

@Component({
  selector: 'app',
  styles: ['.parent {background: lightblue}'],
  template: `

     <div class="parent">
       <h2>Parent</h2>
      <div>Greeting: <input type="text" [value]="greeting"
      (change)="greeting = $event.target.value"></div> //
      <div>User name: <input type="text" [value]="user.name"
      (change)="user.name = $event.target.value"></div>

      <child [greeting]="greeting" [user]="user"></child>
    </div>
  `
})
class AppComponent {
  greeting: string = 'Hello';
```

Объявляет, что объект имеет структурный тип, с целью сохранения изменений. Он используется в методе ngOnChanges()

Входные свойства компонента ChildComponent получают свои значения из компонента AppComponent

Свойство message не имеет аннотации @Input. Мы добавили его для демонстрации того, что изменение его значения не приведет к вызову функции обратного вызова ngOnChanges

Angular вызывает метод ngOnChanges(), когда изменяются привязки к входным свойствам

В родительском компоненте вы изменяете значения свойств greeting и user.name в событии change, которое отправляется, когда эти поля теряют фокус

Значения свойств greeting и user компонента-предка связаны с входными свойствами потомка

```
  user: {name: string} = {name: 'John'};
}

enableProdMode();
@NgModule({
  imports: [ BrowserModule, FormsModule],
  declarations: [ AppComponent, ChildComponent],
  bootstrap: [ AppComponent ]
})
class AppModule { }

platformBrowserDynamic().bootstrapModule(AppModule);
```

Включает производственный режим (см. врезку «Включение производственного режима»)

Когда Angular вызывает метод `ngOnChanges()`, он передает туда значения каждого изменившегося свойства. Значение представлено объектом типа `SimpleChange`, который содержит текущее и предыдущее значение изменившегося входного свойства. Метод `Simple.change.isFirstChange()` позволяет определить, было ли значение свойства изменено, или же свойство получило значение в первый раз. Чтобы аккуратно вывести эти значения, вы используете метод `JSON.stringify()`.

ПРИМЕЧАНИЕ

TypeScript имеет структурную систему типов, поэтому тип аргумента changes метода ngOnChanges() определяется путем включения описания ожидаемых данных. В качестве альтернативы можете объявить интерфейс (например, IChanges {[key: string]: SimpleChange};), и сигнатура функции будет выглядеть так: ngOnChanges(changes: IChanges). Предыдущее объявления свойства user компонента AppComponent — еще один пример использования структурного типа.

Взглянем, приведет ли изменение свойств `greeting` и `user.name` в интерфейсе к вызову метода `ngOnChanges()` компонента-потомка. На рис. 6.11 показан снимок экрана, сделанный после запуска листинга 6.10 с открытой панелью Developer Tools (Инструменты разработчика) в браузере Chrome.

Изначально, когда приложение применило привязку для входных свойств компонента-потомка, они не имели значений. Затем был вызван метод обратного вызова `ngOnChanges()` и предыдущие значения свойств `greeting` и `user` изменились с `{}` на `Hello` и `{name:"John"}` соответственно.

Включение режима коммерческой сборки

На рис. 6.11 вы можете увидеть сообщение о том, что Angular запущен в режиме разработки, в котором выполняются операторы и другие проверки с помощью фреймворка. Один такой оператор проверяет, не приводит ли цикл работы механизма обнаружения изменений к дополнительным изменениям в привязках (например, код не изменяет интерфейс в функциях обратного вызова для жизненного цикла).

Для включения режима коммерческой сборки (prod-конфигурации) вызовите метод `enableProdMode()` в вашем приложении до вызова метода `bootstrap()`. Включение такого режима повысит производительность приложения в браузере.

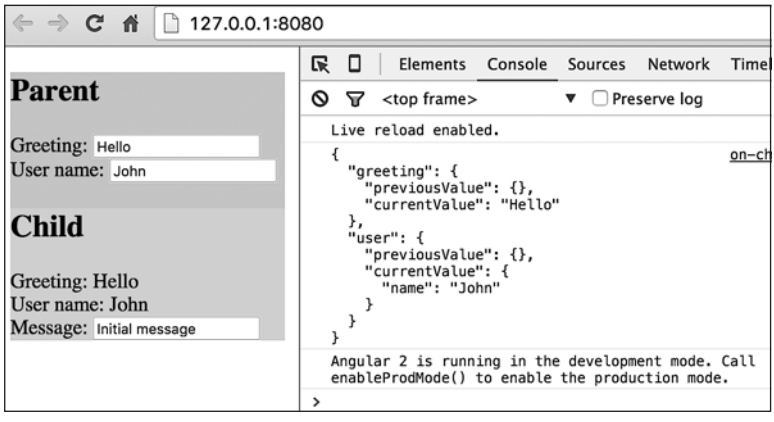


Рис. 6.11. Изначальный вызов ngOnChanges()

Позволим пользователю изменить значения во всех полях ввода. После добавления слова *dear* в поле `Greeting` и переключения фокуса механизм определения изменений Angular обновляет привязку к *неизменяемому* входному свойству потомка, `greeting`. Затем вызывает функцию обратного вызова `ngOnChanges` и выводит предыдущее значение `Hello` и текущее значение `Hello dear`, как показано на рис. 6.12.

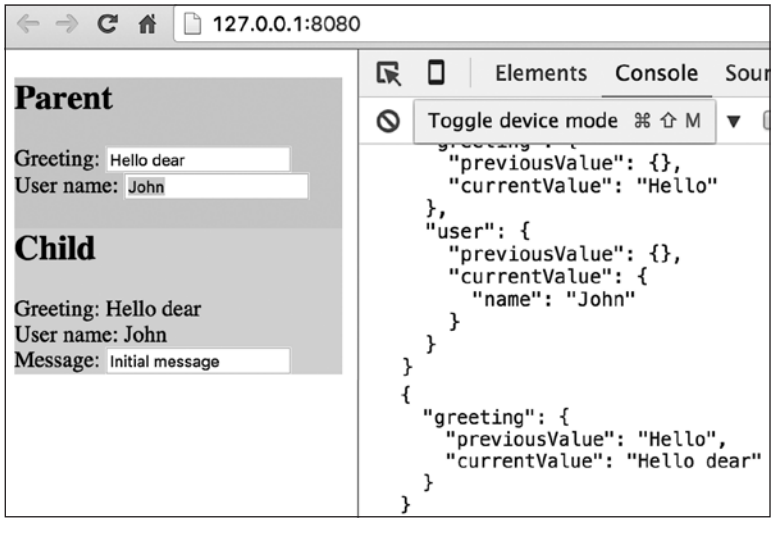


Рис. 6.12. Вызов метода ngOnChanges() после того, как изменилось значение свойства greeting

Теперь предположим, что пользователь добавляет в поле User Name (Имя пользователя) слово `Smith` и убирает оттуда фокус: в консоли не появится новых сообщений, как показано на рис. 6.13, поскольку изменилось только свойство `name` *изменяемого* объекта `user`; ссылка на сам объект не изменится. Это объясняет, почему не был вызван метод `ngOnChanges()`. Изменение значения свойства `message`

компонента `ChildComponent` также не вызывает метод `ngOnChanges()`, поскольку данное свойство не было декорировано аннотацией `@Input`.

Рис. 6.13. Метод ngOnChanges() в этот раз не вызывается

ПРИМЕЧАНИЕ

Несмотря на то что Angular не обновляет привязки для входных свойств, если ссылка на объект не изменилась, механизм обнаружения изменений все еще отлавливает обновления свойств для каждого свойства объекта. Именно поэтому было отрисовано словосочетание John Smith, которое является новым значением поля User Name (Имя пользователя) в компоненте-потомке.

Ранее, в подразделе 6.1.1, вы использовали сеттер, чтобы определить момент, когда изменяется значение параметра ввода. Вместо этого можно задействовать метод `ngOnChanges()`. Существуют ситуации, когда вместо метода `ngOnChanges()` все же лучше применять сеттеры; вы познакомитесь с ними в разделе «Практикум» данной главы.

6.3. Краткий обзор определения изменений

Механизм определения изменений (change-detection, CD) в Angular реализован в файле `zone.js` (также известном как the Zone). Его основное предназначение заключается в том, чтобы синхронизировать значение свойств компонента (модели) и интерфейса. CD инициируется любым асинхронным событием, которое происходит в браузере (пользователь нажал кнопку, данные пришли с сервера, сценарий вызвал функцию `setTimeout()` и т. д.).

Когда CD запускает свой цикл, он проверяет все привязки в шаблоне компонента. Зачем может понадобиться обновлять выражения привязки? Затем, что изменилось одно из свойств элемента.

ПРИМЕЧАНИЕ

Механизм CD переносит изменения, сделанные в свойстве компонента, в интерфейс. CD никогда не изменяет значение свойства компонента.

Вы можете рассматривать приложение как дерево элементов, на вершине которого находится корневой компонент. Когда Angular компилирует шаблоны элемента, каждый компонент получает собственный детектор изменений. Когда CD инициируется Zone, он проходит по всем элементам от корневого до листового, проверяя, нужно ли обновлять интерфейс каждого из них.

В Angular реализованы две стратегии CD: `Default` и `OnPush`. Если все компоненты используют первую стратегию, то Zone проверяет все дерево элементов независимо от того, где произошло изменение. Если какой-то элемент реализует вторую стратегию, то Zone проверяет компонент и его потомков только в случае изменений привязок к его входным свойствам. Чтобы объявить об использовании стратегии `OnPush`, нужно всего лишь добавить в шаблон компонента следующую строку:

```
changeDetection: ChangeDetectionStrategy.OnPush
```

Познакомимся с этими стратегиями с помощью трех компонентов: предка, потомка и «внука», они показаны на рис. 6.14.

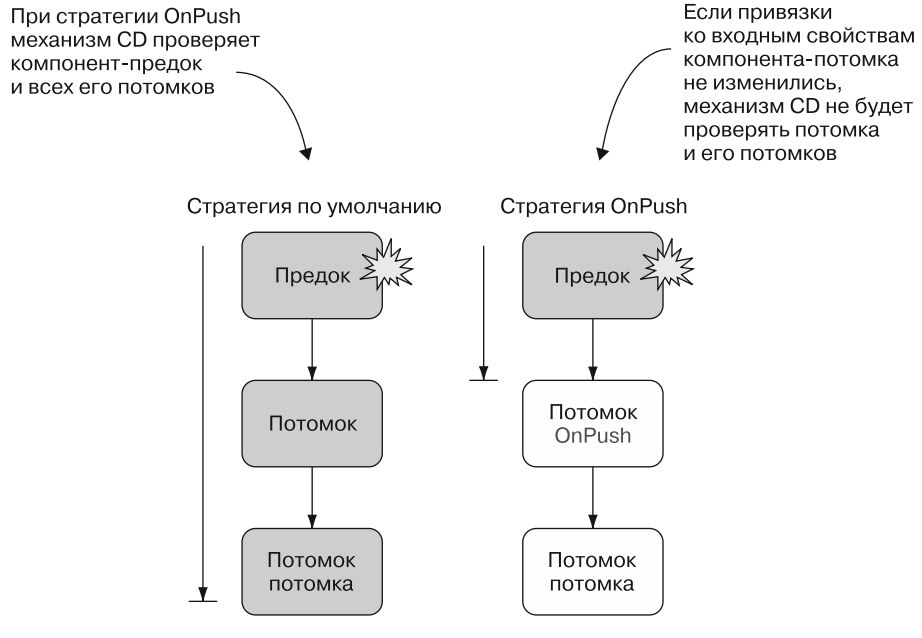


Рис. 6.14. Стратегии определения изменений

Предположим, что изменилось свойство предка. Механизм CD начнет проверять элемент и его потомков. С левой стороны рис. 6.14 показана стратегия CD, используемая по умолчанию: на наличие изменений проверяются все три компонента.

С правой стороны рис. 6.14 показана ситуация, когда для потомка применяется стратегия `OnPush`. Механизм CD начинает свою работу с вершины, но видит, что для потомка объявлена стратегия `OnPush`. Если никакие привязки к входным свойствам данного элемента не изменились, CD не проверяет ни потомка, ни «внука».

На рис. 6.14 показано небольшое приложение, содержащее всего три компонента, но в реальных приложениях могут присутствовать сотни элементов. С помощью стратегии `OnPush` можно отказаться от вызова механизма CD для определенных ветвей дерева.

На рис. 6.15 показан цикл CD, который запустило событие компонента `GrandChild1`. Даже несмотря на то, что это событие произошло в листовом элементе, цикл CD начинает свою работу с вершины. Он выполняется для каждой ветви кроме тех, что происходят из компонента, для которого реализована стратегия `OnPush` и не изменились привязки его входных свойств. Компоненты, исключенные из цикла работы CD, показаны на белом фоне.

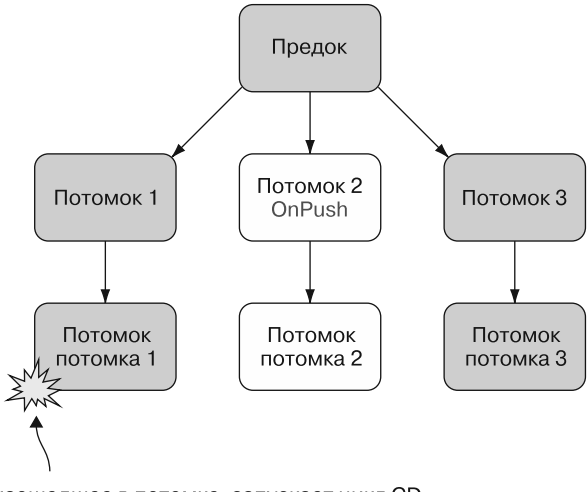


Рис. 6.15. Исключение ветки из цикла CD

Мы кратко рассмотрели механизм CD, который, возможно, является самым сложным модулем Angular. Изучать его более подробно следует только в том случае, если нужно повысить производительность приложения, активно обновляющего интерфейс, например, таблицы, содержащей сотни клеток, чьи значения постоянно изменяются. Для получения более подробной информации об определении изменений в Angular прочтите статью Виктора Савкина (Victor Savkin), которая называется Change Detection in Angular (Определение изменений в Angular) и представлена на https://vsavkin.com/change-detection-in-angular-2-4f216b855d4c.

6.4. Открываем доступ к API компонента-потомка

Вы узнали, как предок может передавать данные своим потомкам с помощью привязок к входным свойствам. Но существуют и другие ситуации, когда предку нужно использовать API, предоставляемый потомком. Мы покажем пример того, как элемент-предок может применять API потомка как для шаблона, так и для кода TypeScript.

Создадим простое приложение, в котором компонент-потомок будет иметь метод `greet()`, вызываемый предком. Чтобы показать разные приемы, предок будет задействовать два экземпляра одного потомка. Эти экземпляры будут иметь разные имена переменных шаблона:

```
<child #child1></child>
<child #child2></child>
```

Теперь можно объявить переменную в коде TypeScript, имеющем аннотацию `@ViewChild`.

Эта аннотация предоставляется Angular для того, чтобы помочь предоставить ссылку на компонент-потомок, и вы будете использовать ее для первого потомка:

```
@ViewChild('child1')
firstChild: ChildComponent;
...
this.firstChild.greet('Child 1');
```

Приведенный код указывает Angular найти компонент-потомок, определяемый переменной шаблона `child1`, и поместить ссылку на данный элемент в переменную `firstChild`.

Чтобы показать другой прием, можно получить доступ ко второму потомку из кода TypeScript, но из шаблона предка. Это делается довольно просто:

```
<button (click)="child2.greet('Child 2')">Invoke greet() on child 2</button>
```

Далее показан весь код, иллюстрирующий оба приема (листинг 6.11).

Листинг 6.11. Содержимое файла exposing-child-api.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule, Component, ViewChild, AfterViewInit } from
➥ '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

@Component({
    selector: 'child',
    template: `<h3>Child</h3>`
})
class ChildComponent {
    greet(name) {
```

```
        console.log(`Hello from ${name}.`);
    }
}

@Component({
    selector: 'app',
    template: `
    <h1>Parent</h1>
    <child #child1></child>
    <child #child2></child>
    <button (click)="child2.greet('Child 2')">Invoke greet() on child 2
    ➥ </button>
  `
})
class AppComponent implements AfterViewInit {
    @ViewChild('child1')
    firstChild: ChildComponent;
    ngAfterViewInit() {
        this.firstChild.greet('Child 1');
    }
}

@NgModule({
    imports:      [ BrowserModule],
    declarations: [ AppComponent, ChildComponent],
    bootstrap:    [ AppComponent ]
})
class AppModule { }
platformBrowserDynamic().bootstrapModule(AppModule);
```

Когда вы запускаете это приложение, оно выводит в консоли сообщение Hello from Child 1. Нажмите кнопку, и приложение выведет сообщение Hello from Child 2., как показано на рис. 6.16.

Рис. 6.16. Получение доступа к API потомка

Обновляем интерфейс с помощью привязок жизненного цикла

В листинге 6.11 используется привязка жизненного цикла `ngAfterViewInit()` для вызова API потомка. Если метод потомка `greet()` не изменяет интерфейс, то данный код отработает как полагается. Однако при попытке изменить интерфейс из метода `greet()` Angular сгенерирует исключение, поскольку интерфейс изменился после вызова метода `ngAfterViewInit()`. Это происходит потому, что данная привязка вызывается в рамках того же цикла событий как для предка, так и для потомка.

Существует два способа решения указанной проблемы. Можно запустить приложение в режиме коммерческой сборки, чтобы Angular не выполнял дополнительные проверки привязок, или же использовать функцию `setTimeout()` для кода, обновляющего интерфейс, для его запуска в следующем цикле событий.

6.5. Практикум: добавление в онлайн-аукцион функциональности для оценивания товаров

В этом разделе вы добавите в приложение-аукцион функциональность для оценивания товаров. В предыдущей версии приложения вы просто отображали рейтинг товара, но теперь хотите дать пользователям возможность оценить продукт. В главе 4 вы создали представление Produce Details (Информация о продукте); здесь же добавите кнопку Leave a Review (Оставить отзыв). Она позволяет пользователям перемещаться к представлению, где они могут присвоить продукту от одной до пяти звезд, а также оставить отзыв. Фрагмент нового представления Produce Details (Информация о продукте) показан на рис. 6.17.

Компонент `StarsComponent` будет иметь входное свойство, которое будет изменяться. Только что добавленное значение рейтинга нужно передать его предку, `ProductItemComponent`.

ПРИМЕЧАНИЕ

Мы возьмем за основу приложение-аукцион, разработанное в главе 5. Если вы предпочитаете увидеть готовый результат этого проекта, то взгляните на исходный код главы 6, расположенный в каталоге auction. В противном случае скопируйте каталог auction из главы 5 в отдельное место. Затем скопируйте файл package.json из каталога auction для главы 6, запустите команду npm install и следуйте инструкциям, предоставленным в данном разделе.

Сделайте следующее.

1. Установите файл определения типа для прокладки ES6. Для этого запустите командную строку и выполните следующую команду:

```
npm install @types/es6-shim --save-dev
```

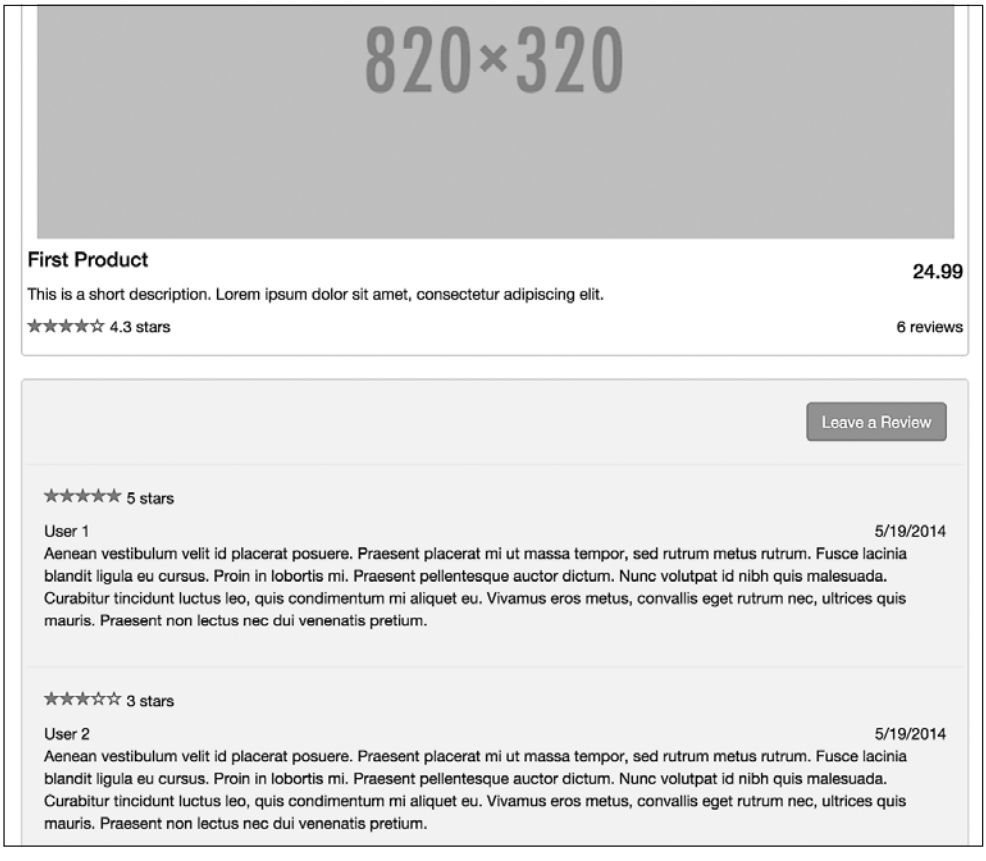


Рис. 6.17. Представление Produce Details (Информация о продукте)

Она установит файл `ES6-shim.d.ts` в каталог `node_modules/@types` и сохранит эту конфигурацию в разделе `devDependencies` файла `package.json file`. Вам необходимо использовать TypeScript 2.0, он знает, как искать файлы с определениями типов в каталоге `@types`.

2. Измените код файла `StarsComponent`. Новая версия должна работать в двух режимах: «только чтение» для отображения звезд на основе данных, полученных от сервиса `ProductService`, и «открыт к записи», чтобы позволить пользователям нажимать звезды для установки нового значения рейтинга.

   На рис. 6.17 показана отрисовка компонента `ProductDetailComponent` (предок), где компонент `StarsComponent` (потомок) находится в режиме «только чтение». Если пользователь нажимает кнопку Leave a Review (Оставить отзыв), то данный режим нужно отключить. Вы добавите входную переменную `readonly` для включения и отключения этого режима.

   Вторая входная переменная, `rating`, предназначена для присвоения рейтинга. Кроме того, вы добавите одну выходную переменную, `ratingChange`, которая

будет отправлять событие, содержащее новую оценку; оно будет использоваться компонентом-предком для пересчета среднего рейтинга.

Когда пользователь нажмет одну из звезд, будет вызван метод `fillStarsWithColor()`, который присвоит значение переменной `rating` и отправит ее значение с помощью события. Измените код файла `stars.ts`, чтобы он выглядел так, как показано в листинге 6.12.

Листинг 6.12. Обновленный файл stars.ts

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

  @Component({
    selector: 'auction-stars',
    styles: [`.starrating { color: #d17581; }`],
    templateUrl: 'app/components/stars/stars.html'
  })
  export default class StarsComponent {
    private _rating: number;
    private stars: boolean[];
    private maxStars: number =5;
    @Input() readonly: boolean = true;
    @Input() get rating(): number {
      return this._rating;
    }

  set rating(value: number) {
    this._rating = value || 0;
    this.stars = Array(this.maxStars).fill(true, 0, this.rating);
  }

  @Output() ratingChange: EventEmitter<number> = new EventEmitter();

  fillStarsWithColor(index) {
    if (!this.readonly) {
      this.rating = index + 1;
      this.ratingChange.emit(this.rating);
    }
  }
}
```

Вы задействуете сеттер для установки значения рейтинга. Этот сеттер может быть вызван либо из компонента `StarsComponent` (для отрисовки существующего рейтинга), либо из его предка (когда пользователь нажимает звезду). В данном приложении использование `ngOnChanges()` не сработает, поскольку он будет вызван предком всего раз при создании компонента `StarsComponent`.

Обратите внимание на применение метода `fill()` из ES6 в сеттере `rating()`. Вы заполняете массив `stars` значениями `true`, начиная с нулевого элемента и заканчивая элементом с номером, совпадающим со значением рейтинга. Чтобы заполнить звезду цветом, сохраняете значение `true`; для пустых звезд сохраняете `false`.

3. Измените шаблон компонента `StarsComponent` в файле `stars.html`, как это показано в листинге 6.13. С помощью директивы `ngFor` вы проходите в цикле по массиву `stars`, в котором хранятся булевы значения. Для пустых и закрашенных звезд вы будете использовать готовые изображения, поставляемые с библиотекой Bootstrap (см. https://getbootstrap.com/docs/3.3/components/). В зависимости от значения элемента массива вы будете отрисовывать либо закрашенную, либо пустую звезду. Когда пользователь нажимает на звезду, вы передаете ее индекс функции `fillStarsWithColor()`.

Листинг 6.13. Переработанный файл stars.html

```
<p>
  <span *ngFor="let star of stars; let i = index"
        class="starrating glyphicon glyphicon-star"
        [class.glyphicon-star-empty]="!star"
        (click)="fillStarsWithColor(i)">
  </span>
  <span *ngIf="rating">{{rating | number:'.0-2'}} stars</span>
</p>
```

Канал `number` форматирует значение рейтинга так, чтобы после десятичной запятой отображались два разряда.

4. Измените шаблон компонента `ProductDetailComponent`. Данный элемент имеет кнопку Leave a Review (Оставить отзыв), позволяющую оценить продукт и оставить отзыв о нем. Ее нажатие изменит видимость элемента `<div>`, который дает пользователям возможность нажимать на звезды и оставлять отзыв, как показано на рис. 6.18.

Рис. 6.18. Представление Leave a Review (Оставить отзыв)

Здесь компонент `StarsComponent` можно редактировать, и пользователь может дать выбранному продукту до пяти звезд. Шаблон, реализующий представление, показанное на рис. 6.18, будет выглядеть так:

```
<div [hidden]="isReviewHidden">
    <div><auction-stars [(rating)]="newRating"
      [readonly]="false" class="large"></auction-stars></div>
    <div><textarea [(ngModel)]="newComment"></textarea></div>
    <div><button (click)="addReview()" class="btn">Add review
    ➥ </button></div>
</div>
```

Режим «только чтение» отключен. Обратите внимание на то, что в двух местах используется двухсторонняя привязка: `[(rating)]` и `[(ngModel)]`. Ранее в данной главе вы рассматривали применение директивы `ngModel` в двухсторонней привязке, но если у вас есть входное свойство (например, `rating`) и выходное, имеющее такое же имя и суффикс `Change` (например, `ratingChange`), то можете задействовать для них синтаксис `[()]`.

Кнопка Leave a Review (Оставить отзыв) изменяет видимость элемента `<div>`. Это реализуется так:

```
<button (click)="isReviewHidden = !isReviewHidden"
                class="btn btn-success btn-green">Leave a Review</button>
```

Замените содержимое файла `product-detail.html` на приведенное ниже (листинг 6.14).

Листинг 6.14. Измененный файл product-detail.html

```
<div class="thumbnail">
    <img src="http://placehold.it/820x320">
    <div>
        <h4 class="pull-right">{{ product.price }}</h4>
        <h4>{{ product.title }}</h4>
        <p>{{ product.description }}</p>
    </div>
    <div class="ratings">
        <p class="pull-right">{{ reviews.length }} reviews</p>
        <p><auction-stars [rating]="product.rating" ></auction-stars></p>
    </div>
</div>

<div class="well" id="reviews-anchor">
    <div class="row">
        <div class="col-md-12"></div>
    </div>
    <div class="text-right">
        <button (click)="isReviewHidden = !isReviewHidden"
                class="btn btn-success btn-green">Leave a Review</button>
    </div>

    <div [hidden]="isReviewHidden">
        <div><auction-stars [(rating)]="newRating"
          [readonly]="false" class="large"></auction-stars></div>
        <div><textarea [(ngModel)]="newComment"></textarea></div>
        <div><button (click)="addReview()" class="btn">Add review</button>
        ➥ </div>
    </div>

    <div class="row" *ngFor="#review of reviews">
        <hr>
```

```
        <div class="col-md-12">
            <auction-stars [rating]="review.rating"></auction-stars>
            <span>{{ review.user }}</span>
            <span class="pull-
              right">{{ review.timestamp | date: 'shortDate' }}</span>
            <p>{{ review.comment }}</p>
        </div>
    </div>
</div>
```

После ввода текста отзыва и оценки продукта пользователь нажимает кнопку Add Review (Добавить отзыв), которая вызывает для компонента метод `addReview()`. Реализуем его с помощью TypeScript.

5. Измените файл `product-detail.ts`. Для добавления отзыва нужно делать следующее: отправить новый отзыв на сервер и пересчитать средний рейтинг продукта в интерфейсе. Вы проделаете последнее действие, еще не реализовав коммуникацию с сервером; отзыв мы будем размещать в консоли браузера. Далее вы добавите новый отзыв в массив существующих отзывов. В следующем фрагменте кода компонента `ProductDetailComponent` реализуется эта функциональность:

```
addReview() {
  let review = new Review(0, this.product.id, new Date(), 'Anonymous',
      this.newRating, this.newComment);
  console.log("Adding review " + JSON.stringify(review));
  this.reviews = [...this.reviews, review];

  this.product.rating = this.averageRating(this.reviews);
  this.resetForm();
}

averageRating(reviews: Review[]) {
  let sum = reviews.reduce((average, review) => average + review.rating, 0);
  return sum / reviews.length;
}
```

После создания нового экземпляра объекта класса `Review` вам нужно добавить его в массив `reviews`. Оператор расширения позволяет записать его элегантным образом:

```
this.reviews = [...this.reviews, review];
```

Массив `reviews` получает значения всех существующих элементов (`...this.reviews`), а также нового (`review`). Пересчитанное среднее значение присваивается свойству `rating`, значение которого попадает в интерфейс с помощью привязки.

Что нам осталось сделать? Заменим содержимое файла `product-detail.ts` на следующий код (листинг 6.15), и это упражнение закончится!

Листинг 6.15. Измененный файл product-detail.ts

```
import {Component} from '@angular/core';
import {ActivatedRoute} from '@angular/router';
import {Product, Review, ProductService} from
➥ '../../services/product-service';
import StarsComponent from '../stars/stars';

4@Component({
  selector: 'auction-product-page',
  styles: ['auction-stars.large {font-size: 24px;}'],
  templateUrl: 'app/components/product-detail/product-detail.html'
})

export default class ProductDetailComponent {
  product: Product;
  reviews: Review[];

  newComment: string;
  newRating: number;

  isReviewHidden: boolean = true;

  constructor(route: ActivatedRoute, productService: ProductService) {

    let prodId: number = parseInt(route.snapshot.params['productId']);
    this.product = productService.getProductById(prodId);
    this.reviews = productService.getReviewsForProduct(this.product.id);
  }

  addReview() {
    let review = new Review(0, this.product.id, new Date(), 'Anonymous',
        this.newRating, this.newComment);
    console.log("Adding review " + JSON.stringify(review));
    this.reviews = [...this.reviews, review];
    this.product.rating = this.averageRating(this.reviews);

    this.resetForm();
  }

  averageRating(reviews: Review[]) {
    let sum = reviews.reduce((average, review) => average + review.rating,0);
    return sum / reviews.length;
  }

  resetForm() {
    this.newRating = 0;
    this.newComment = null;
    this.isReviewHidden = true;
  }
}
```

6.6. Резюме

Любое Angular-приложение представляет собой иерархию компонентов, которым нужно коммуницировать друг с другом. В этой главе рассматривались разные способы организации подобной коммуникации. Привязки к входным свойствам элемента и отправка событий с помощью выходных свойств позволяют создавать слабо связанные компоненты. Используя механизм определения изменений, Angular перехватывает изменения в свойствах элемента для гарантии того, что его привязки обновляются.

Каждый компонент проходит через определенную серию событий в течение своего жизненного цикла. Angular предоставляет несколько привязок жизненного цикла, где можно писать код для перехвата данных событий и применять пользовательскую логику.

Вот основные выводы этой главы.

- Компоненты-предки и компоненты-потомки не должны получать прямой доступ к содержимому друг друга, но им следует общаться с помощью входных и выходных свойств.
- Элемент может отправлять пользовательские события, используя свои выходные свойства, и эти события могут нести полезную нагрузку, характерную для приложения.
- Коммуникация между несвязанными компонентами может быть выстроена с помощью шаблона проектирования «Посредник».
- Предок может передавать один или несколько фрагментов шаблона потомкам во время выполнения.
- Каждый компонент Angular позволяет перехватывать основные события жизненного цикла элемента и обрабатывать их, задействуя собственный код.
- Механизм определения изменений Angular автоматически отслеживает изменения свойств компонента и соответствующим образом обновляет интерфейс.
- Можно отметить выбранные ветви дерева компонентов приложения, чтобы исключить их из процесса обнаружения изменений.

7 Работа с формами

В этой главе:

- Angular Forms API (`NgModel`, `FormControl`, `FormGroup`, директивы форм, `FormBuilder`);
- работа с шаблон-ориентированными формами;
- работа с реактивными формами;
- валидация форм.

Angular обеспечивает поддержку огромного количества форм данных. Это выходит за рамки обычной привязки данных, поскольку поля форм считаются объектами первого класса, а над данными формы у вас есть полный контроль.

Данная глава начнется с демонстрации того, как можно реализовать пример формы регистрации пользователей на чистом HTML. Работая над этой формой, мы вкратце обсудим стандартные формы HTML и их недостатки. Затем покажем, какие возможности предлагает Forms API, и рассмотрим шаблон-ориентированный и реактивный подходы к созданию форм в Angular.

После изучения основ вы переработаете исходную версию регистрационной формы пользователя так, чтобы в ней использовался шаблон-ориентированный подход, и мы обсудим его плюсы и минусы. Затем сделаем то же самое с реактивным подходом. После этого обсудим валидацию форм. В конце главы вы примените новые знания при написании приложения онлайн-аукциона и начнете реализовывать один из его компонентов — форму поиска.

Шаблон-ориентированный подход против реактивного подхода

При *шаблон-ориентированном* подходе формы полностью запрограммированы в шаблоне компонента. Он определяет структуру формы, формат ее полей и правила валидации.

Напротив, используя реактивный подход, вы программно создаете модели форм в коде (в данном случае в коде TypeScript). Шаблон может быть либо определен статически и привязан к существующей модели форм, либо сгенерирован динамически, основываясь на модели.

К концу главы вы познакомитесь с Angular Forms API, с различными способами работы с формами, а также с выполнением валидации данных.

7.1. Обзор форм HTML

В HTML предоставляются основные функции для отображения форм, валидации введенных значений и отправки данных на сервер. Но формы HTML могут быть недостаточно хороши для реальных бизнес-приложений, для которых требуется способ программно обработать введенные данные, применить пользовательские правила валидации, отобразить удобные для пользователя сообщения об ошибках, преобразовать формат введенных данных и выбрать способ отправки данных на сервер. Для бизнес-приложений одним из самых важных факторов при выборе веб-фреймворка является то, насколько хорошо он работает с формами.

В этом разделе мы рассмотрим стандартную функциональность HTML для работы с формами на примере формы регистрации пользователей и определим список требований к современным веб-приложениям, выполнение которых позволит им соответствовать ожиданиям пользователей. Мы также обсудим функциональность для работы с формами, предоставляемую Angular.

7.1.1. Стандартная функциональность браузера

Вы, наверное, задаетесь вопросом, что еще нужно от фреймворка, помимо привязки данных, если HTML уже позволяет проверять и отправлять формы. Чтобы ответить на этот вопрос, взглянем на форму HTML, которая использует только стандартную функциональность браузера (листинг 7.1).

Листинг 7.1. Форма регистрации пользователя, написанная на простом HTML

```
<form action="/register" method="POST">
  <div>Username:         <input type="text"></div>
  <div>SSN:              <input type="text"></div>
  <div>Password:         <input type="password"></div>
  <div>Confirm password: <input type="password"></div>
  <button type="submit">Submit</button>
</form>
```

Эта форма содержит кнопку и четыре поля ввода: имя пользователя, номер социального страхования (Social Security Number, SSN), пароль и подтверждение пароля. Пользователи могут вводить абсолютно любые значения: вводимые данные здесь не проверяются. Когда пользователь нажимает кнопку Submit (Отправить), значения в форме отправляются в конечную точку сервера `/register` с помощью HTTP-запроса `POST`, после чего страница обновляется.

Поведение стандартных форм HTML не всегда подходит для одностраничных приложений, для которых чаще всего требуется следующая функциональность.

- Правила валидации должны быть установлены для отдельных полей ввода.
- Сообщения об ошибке должны отображаться рядом с теми полями ввода, с которыми возникли проблемы.
- Зависимые поля нужно проверять одновременно. Данная форма имеет поля для ввода и подтверждения пароля, поэтому при изменении значения в одном из них нужно заново выполнить валидацию обоих полей.

- Приложение должно контролировать значения, которые отправляются на сервер. Когда пользователь нажимает кнопку Submit (Отправить), приложение должно вызвать функцию — обработчик событий, чтобы передать значения формы. Приложение может проверять значения или изменять их формат перед тем, как создать запрос на отправку.
- Приложение должно решить, как отправить данные на сервер: с помощью обычного HTTP-запроса, запроса Ajax или сообщения WebSocket.

Атрибуты валидации HTML и семантические типы входных данных частично удовлетворяют первым двум требованиям.

Атрибуты валидации HTML

Существует несколько стандартных атрибутов валидации, которые позволяют проверять отдельные поля ввода: `required`, `pattern`, `maxlength`, `min`, `max`, `step` и т. д. Например, можно указать, что поле `username` является обязательным и его значение должно содержать только буквы и цифры.

```
<input id="username" type="text" required pattern="[a-zA-Z0-9]+">
```

Здесь используется регулярное выражение `[a-zA-Z0-9]+`, чтобы ограничить диапазон вводимых значений для этого поля. Когда пользователь нажимает кнопку Submit (Отправить), форма будет проверена до того, как сформируется запрос на отправку. На рис. 7.1 показано стандартное сообщение об ошибке, отображенное в браузере Chrome, когда значения в поле `username` не соответствуют заданному шаблону.

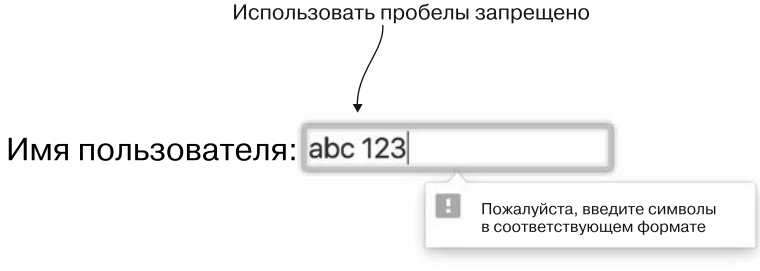


Рис. 7.1. Сообщение об ошибке валидации

У этого сообщения есть несколько недостатков:

- оно слишком расплывчатое и не помогает пользователю идентифицировать и решить проблему;
- как только поле ввода теряет фокус, сообщение об ошибке исчезает;
- такой формат сообщения не будет соответствовать другим стилям в приложении.

Приведенное поле ввода не позволяет пользователю вводить недопустимые значения, но при этом вы не можете сделать сайт более удобным, предоставив возможность проверять данные на стороне клиента.

Семантические типы входных данных

HTML поддерживает множество типов элементов ввода данных: `text`, `number`, `url`, `email` и т. д. Выбор правильного типа поля формы может помешать пользователю вводить недопустимые значения. И, хотя такой выбор делает сайт более удобным, этого все равно недостаточно для удовлетворения нужд вашего приложения, связанных с проверкой данных.

Рассмотрим поле для ввода ZIP-кода (почтовый индекс в США). Может возникнуть соблазн использовать элемент ввода `number`, так как ZIP-код представляет собой числовое значение (по крайней мере в Соединенных Штатах). Чтобы значения оставались в определенном диапазоне, можно задействовать атрибуты `min` и `max`. Например, для пятизначного ZIP-кода подойдет следующая разметка:

```
<input id="zipcode" type="number" min="10000" max="99999">
```

Но не каждое пятизначное число является действительным ZIP-кодом. В более сложном примере может понадобиться объявить допустимыми лишь некоторые ZIP-коды в зависимости от того, в каком штате находится пользователь.

Для решения реальных задач нужна более продвинутая поддержка форм, которую может предоставить фреймворк приложения. Посмотрим, что в данном случае предлагает Angular.

7.1.2. Forms API в Angular

Существует два подхода в работе с формами в Angular: *шаблон-ориентированный* и *реактивный*. Для каждого из них в Angular предоставляется отдельный API (набор директив и классов TypeScript).

В шаблон-ориентированном подходе модель формы определяется шаблоном компонента с помощью директив. Поскольку при определении модели формы вы ограничены синтаксисом HTML, этот подход годится только для простых сценариев.

Для сложных форм больше подходит реактивный подход. Используя его, вы создаете базовую структуру данных непосредственно в коде (а не в шаблоне). После создания модели связываете элементы шаблона HTML с моделью, задействуя специальные директивы с префиксом `form*`. В отличие от шаблон-ориентированных реактивные формы можно протестировать без участия браузера.

Выделим несколько важных концепций, чтобы в дальнейшем прояснить разницу между шаблон-ориентированными и реактивными формами.

- В обоих типах форм есть модель, являющаяся базовой структурой данных, хранящей данные формы. В шаблон-ориентированном подходе Angular неявно создает модель на основе директив, которые вы прикрепляете к элементам шаблона. В реактивном подходе вы создаете модель явно и потом привязываете элементы шаблона HTML к данной модели.
- Модель *не является* произвольным объектом. Это объект, который был создан с использованием классов, определенных в модуле `@angular/forms`: `FormControl`,

`FormGroup` и `FormArray`. В шаблон-ориентированном подходе вы не работаете с указанными классами напрямую, в то время как, используя реактивный подход, вы явным образом создаете экземпляры этих классов.

- Применение реактивного подхода не освобождает от написания шаблона HTML. Angular не будет генерировать представление за вас.

Включение поддержки Forms API

Оба типа форм — шаблон-ориентированные и реактивные — должны быть явно включены до того, как вы начнете их использовать. Чтобы включить шаблон-ориентированные формы, добавьте модуль `FormsModule` из `@angular/forms` в список `imports` директивы `NgModule`, которая задействует Forms API. Для реактивных форм применяйте модуль `ReactiveFormsModule`. Вот как это делается:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { platformBrowserDynamic } from
➥ '@angular/platform-browser-dynamic';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';

@NgModule({
  imports : [ BrowserModule, FormsModule, ReactiveFormsModule ],
  declarations: [ AppComponent ],
  bootstrap : [ AppComponent ]
})
class AppModule {}

platformBrowserDynamic().bootstrapModule(AppModule);
```

Оба модуля форм могут быть импортированы в одном модуле приложения

Мы не будем повторять данный код для каждого примера в данной главе — все они предполагают, что эти модули импортированы. Все загружаемые фрагменты кода для нашей книги импортируют модули в `AppModule`.

7.2. Шаблон-ориентированные формы

Как мы говорили ранее, чтобы определить модель в шаблон-ориентированном подходе, можно использовать только директивы. Но какие именно? Эти директивы поставляются с модулем `FormsModule`: `NgModel`, `NgModelGroup` и `NgForm`.

В главе 5 мы обсудили, каким образом можно применять директиву `NgModel` для двухсторонней привязки данных. Но в Forms API у нее совершенно другая роль: она отмечает элемент HTML, который должен стать частью модели формы. Несмотря на то, что эти две роли различны, они не конфликтуют друг с другом и могут быть безопасно использованы в одном элементе HTML. Примеры мы рассмотрим ниже в данном разделе. Кратко рассмотрим упомянутые директивы, а потом применим шаблон-ориентированный подход к нашей форме регистрации.

7.2.1. Обзор директив

Здесь мы дадим краткое описание трех основных директив из модуля `FormsModule`: `NgModel`, `NgModelGroup` и `NgForm`. Мы покажем, каким образом их можно использовать в шаблоне, и выделим их наиболее важные особенности.

NgForm

Это директива, представляющая собой форму целиком. Она автоматически прикрепляется к каждому элементу `<form>`. Косвенным образом создает экземпляр класса `FormGroup`, который представляет данную модель и хранит данные формы (подробнее о `FormGroup` вы узнаете далее в этой главе). Директива `NgForm` автоматически распознает все элементы-потомки HTML, отмеченные директивой `NgModel`, и добавляет их значения в модель формы.

Директива `NgForm` имеет несколько селекторов, которые можно использовать для того, чтобы прикрепить ее к другим элементам, а не только `<form>`:

```
<div ngForm></div>    ← Селектор атрибута
<ngForm></ngForm>  ←┐
                    └ Селектор элемента
```

Этот синтаксис пригодится в случае использования фреймворка CSS, который требует, чтобы у элементов HTML была определенная структура и элемент `<form>` не может быть задействован.

Если вы хотите, чтобы Angular исключил какой-то конкретный элемент `<form>` из обработки, то используйте атрибут `ngNoForm`:

```
<form ngNoForm></form>
```

Атрибут `ngNoForm` предотвращает создание экземпляра директивы `NgForm` и его прикрепление к элементу `<form>`.

Директива `NgForm` имеет свойство `exportAs`, объявленное в аннотации `@Directive`, позволяющее использовать значение этого свойства для создания локальной переменной шаблона, которая ссылается на экземпляр директивы `NgForm`:

```
<form #f="ngForm"></form>
<pre>{{ f.value | json }}</pre>
```

Сначала вы указываете `ngForm` в качестве значения свойства `exportAs` директивы `NgForm`; переменная `f` указывает на экземпляр директивы `NgForm`, прикрепленный к элементу `<form>`. Затем вы можете использовать переменную `f` для получения доступа к членам объекта директивы `NgForm`. Один из этих членов — `value` — представляет текущее значение всех полей формы в качестве объекта JavaScript. Вы можете пропустить его через стандартный канал `json`, чтобы отобразить значение формы на странице.

Директива `NgForm` перехватывает стандартное событие формы HTML `submit` и предотвращает автоматическую отправку формы. Вместо этого она генерирует пользовательское событие `ngSubmit`:

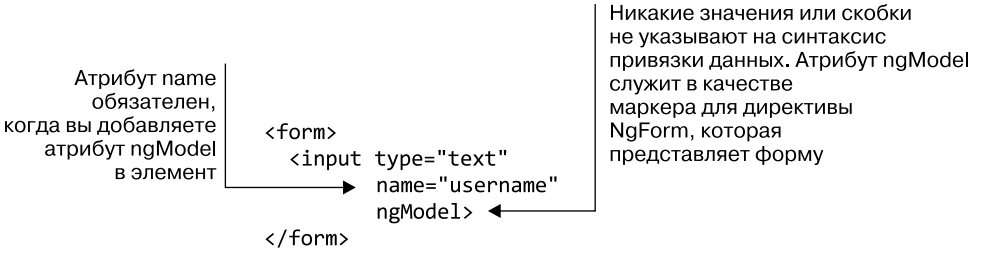
```
<form #f="ngForm" (ngSubmit)="onSubmit(f.value)"></form>
```

Данный код подписывается на событие `ngSubmit`, используя синтаксис привязки событий. Элемент `onSubmit` — произвольное имя метода, определенного в компоненте, вызываемого в тот момент, когда генерируется событие `ngSubmit`. Для передачи всех значений формы в качестве аргумента этого метода задействуйте переменную `f`, чтобы получить доступ к свойству `value` директивы `NgForm`.

NgModel

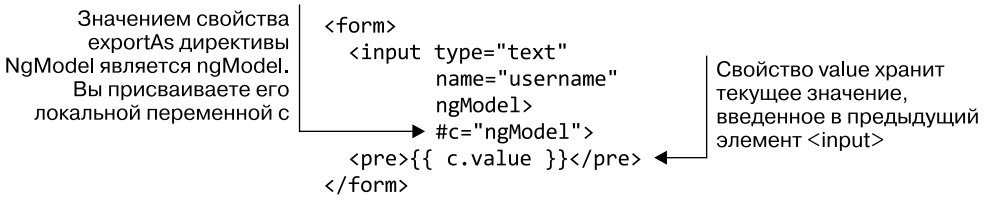
В контексте Forms API данная директива является отдельным полем на форме. Она неявно создает экземпляр класса `FormControl`, который представляет собой данную модель и хранит данные полей ввода (подробнее о `FormControl` вы узнаете далее в этой главе).

Вы прикрепляете объект `FormControl` к элементу HTML с помощью атрибута `ngModel`. Обратите внимание: Forms API не требует, чтобы атрибуту `ngModel` было присвоено какое-либо значение, а также заключения атрибута в скобки:



Свойство `NgForm.value` указывает на объект JavaScript, содержащий значения всех полей формы. Значение атрибута поля `name` становится именем свойства объекта JavaScript в `NgForm.value`.

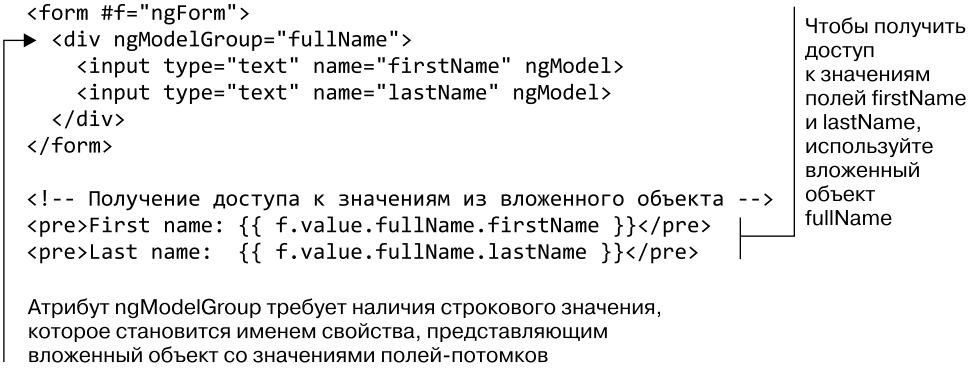
Как и `NgForm`, директива `NgModel` имеет свойство `exportAs`, поэтому можно создавать переменную в шаблоне, которая будет ссылаться на экземпляр директивы `NgModel` и на ее свойство `value`:



NgModelGroup

Данная директива представляет собой часть формы и позволяет группировать ее поля. Подобно директиве `NgForm`, неявно создает экземпляр класса `FormGroup`. По сути, `NgModelGroup` создает вложенный объект в объекте, который хранится в свойстве `NgForm.value`. Все поля — потомки объекта класса `NgModelGroup` — становятся свойствами вложенного объекта.

Вот как можно это использовать:



7.2.2. Доработка формы HTML

Переработаем код примера формы регистрации пользователя, показанный в листинге 7.1. Выше была приведена форма, написанная на простом HTML, в которой не применялись какие-либо функции Angular. Теперь вы превратите ее в компонент Angular, добавите логику валидации и включите программную обработку события `submit`. Начнем с переработки кода шаблона, а потом перейдем к части, написанной на TypeScript. В первую очередь модифицируем элемент `<form>` (листинг 7.2).

Листинг 7.2. Форма с поддержкой Angular

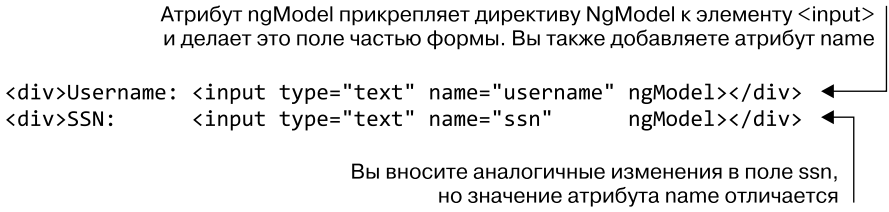
```
<form #f="ngForm" (ngSubmit)="onSubmit(f.value)">
        <!-- Здесь будут располагаться поля форм -->
</form>
```

Вы объявляете локальную переменную `f`, которая указывает на объект директивы `NgForm`, прикрепленный к элементу `<form>`. Вам необходима эта переменная, чтобы получить доступ к таким свойствам формы, как `value` и `valid`, и проверить, имеет ли форма ошибки определенного типа.

Вы также настраиваете обработчик для события `ngSubmit`, которое генерируется директивой `NgForm`. Вы не хотите отслеживать стандартное событие `submit`, и `NgForm` перехватывает событие `submit` и останавливает его распространение. Так предотвращается автоматическая отправка формы на сервер, приводящая к перезагрузке страницы. Вместо этого директива `NgForm` генерирует свое собственное событие `ngSubmit`.

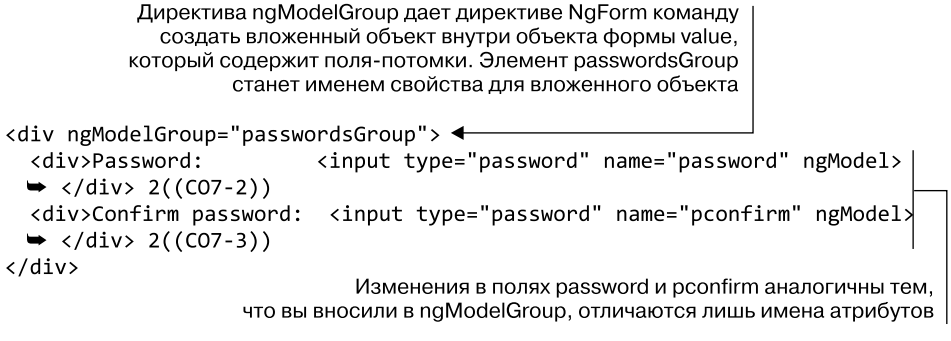
Метод `onSubmit()` является обработчиком событий. Он определяется как метод экземпляра компонента. В шаблон-ориентированных формах метод `onSubmit()` принимает один аргумент: значение формы, которая является простым объектом JavaScript, содержащим значения всех полей формы. Затем вы изменяете поля `username` и `ssn` (листинг 7.3).

Листинг 7.3. Модифицированные поля username и ssn



Теперь изменим поля для ввода пароля (листинг 7.4). Поскольку они связаны друг с другом и представляют одинаковые значения, вполне естественно объединить их в группу. Кроме того, будет удобно рассматривать оба пароля как один объект, когда вы будете реализовывать валидацию формы далее в этой главе.

Листинг 7.4. Модифицированные поля ввода паролей



Кнопка Submit (Отправить) остается единственным элементом HTML в шаблоне, она не изменилась по сравнению с кнопкой, использованной в HTML-версии формы:

```
<button type="submit">Submit</button>
```

Теперь, когда вы закончили перерабатывать код шаблона, обернем его в компонент. Ниже приведен код компонента (листинг 7.5).

Листинг 7.5. Компонент формы HTML

```
@Component({
  selector: 'app',
  template: `...`
})
class AppComponent {
  onSubmit(formValue: any) {
    console.log(formValue);
  }
}
```

Мы не включали содержимое шаблона, чтобы листинг оставался кратким, но здесь должна находиться его переработанная версия, описанная ранее.

Обработчик событий `onSubmit()` принимает единственный аргумент — значение формы. Как вы могли заметить, обработчик не использует API, характерный для Angular. В зависимости от значения флага валидации вы можете решить, следует ли отправлять `formValue` на сервер. В этом примере вы выводите его в консоли.

На рис. 7.2 отображен наш образец регистрационной формы, к которой были применены директивы формы. Каждая директива обведена, чтобы вы могли видеть, из чего состоит форма. Полное рабочее приложение, иллюстрирующее, как использовать директивы форм, находится в файле `01_template-driven.ts` в коде, поставляемом вместе с книгой.

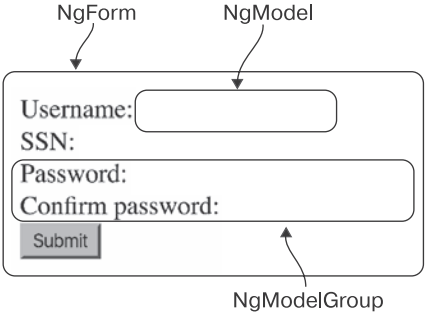


Рис. 7.2. Директивы для форм на примере регистрационной формы

7.3. Реактивные формы

В отличие от шаблон-ориентированного подхода процесс создания реактивной формы состоит из двух этапов. Сначала необходимо создать модель программно в коде, а затем привязать элементы HTML к данной модели с помощью директив в шаблоне. Начнем с первого этапа — создания модели.

7.3.1. Модель формы

Модель формы — базовая структура данных, которая содержит данные формы. Она создается из специальных классов, определенных в модуле `@angular/forms`: `FormControl`, `FormGroup` и `FormArray`.

FormControl

Класс `FormControl` — неделимый элемент формы. Обычно он соответствует одному элементу `<input>`, но может представлять собой и более сложный компонент пользовательского интерфейса, например календарь или слайдер. Объект класса `FormControl` содержит текущее значение соответствующего элемента HTML, информацию о статусе валидации элемента, а также сведения о том, был ли он изменен.

Вот как вы можете создать элемент управления:

```
let username = new FormControl('initial value');
```

Передайте начальное значение элемента управления в качестве первого аргумента конструктора

FormGroup

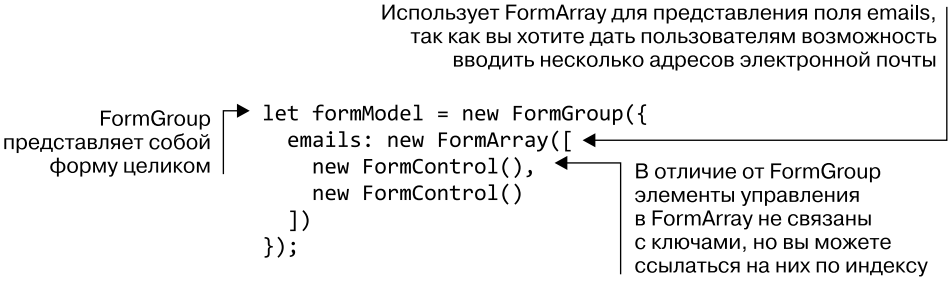
Обычно представляет собой часть формы и является набором из нескольких `FormControl`. Собирает вместе значения и статусы каждого `FormControl` в группе. Если один из элементов управления в группе имеет недопустимые значения, то считается, что недопустимые значения имеет вся группа. Это удобно для управления связанными полями формы. Модель `FormGroup` также используется для представления формы целиком. Например, если диапазон дат представлен двумя полями ввода `date`, то они могут быть собраны в одну группу, чтобы получить диапазон дат как одно значение и отобразить ошибку, когда одна из введенных дат имеет недопустимое значение.

Вот как можно создать группу, которая объединяет элементы управления `from` и `to`:

```
let formModel = new FormGroup({
  from: new FormControl(),
  to :  new FormControl()
});
```

FormArray

Аналогична `FormGroup`, но имеет переменную длину. В отличие от `FormGroup`, которая представляет собой форму целиком или фиксированное подмножество полей формы, модель `FormArray` обычно является коллекцией полей, способной увеличиваться в размерах. Например, можно использовать `FormArray` для того, чтобы позволить пользователям вводить произвольное количество адресов электронной почты. Ниже приведена модель, которая может лечь в основу подобной формы:



7.3.2. Директивы форм

В реактивном подходе используется совершенно другой набор директив, отличный от шаблон-ориентированных форм. Директивы для реактивных форм поставляются вместе с модулем `ReactiveFormsModule` (см. раздел 7.2).

Имена всех реактивных директив начинаются со строки `form*`, поэтому можно легко отличить реактивную форму от шаблон-ориентированной, просто взглянув на шаблон. Реактивные директивы нельзя экспортировать; это значит, что вы

не можете создать переменную в шаблоне, которая ссылается на экземпляр директивы. Так сделано специально, чтобы четко разделить два подхода. В шаблон-ориентированных формах вы не получаете доступ к классам модели, а в реактивных формах не можете работать с моделью в шаблоне.

В табл. 7.1 показано, каким образом классы модели соответствуют директивам формы. В первой колонке представлен список классов, которые мы рассмотрели в предыдущем разделе. Во второй даны директивы, привязывающие элемент DOM к экземпляру класса модели с помощью синтаксиса привязки свойств. Как вы можете заметить, `FormArray` не может использоваться с привязкой свойств. В третьей колонке представлен список директив, связывающих элемент DOM с классом модели, задействуя имя. Они могут применяться только в директиве `formGroup`.

Таблица 7.1. Соответствие классов модели директивам форм

Класс модели	**Директивы форм**	
FormGroup	formGroup	formGroupName
FormControl	formControl	formControlName
FormArray	—	formArrayName

Рассмотрим директивы форм.

formGroup

Зачастую привязывает экземпляр объекта класса `FormGroup`, который представляет собой модель формы целиком, к элементу DOM высшего уровня вашей формы; обычно это элемент `<form>`. Все директивы, прикрепленные к элементам-потомкам DOM, будут находиться в области видимости `formGroup` и могут привязывать экземпляры моделей с помощью имени.

Чтобы использовать директиву `formGroup`, сначала создайте в компоненте объект класса `FormGroup`:

```
@Component(...)
class FormComponent {
  formModel: FormGroup = new FormGroup({});
}
```

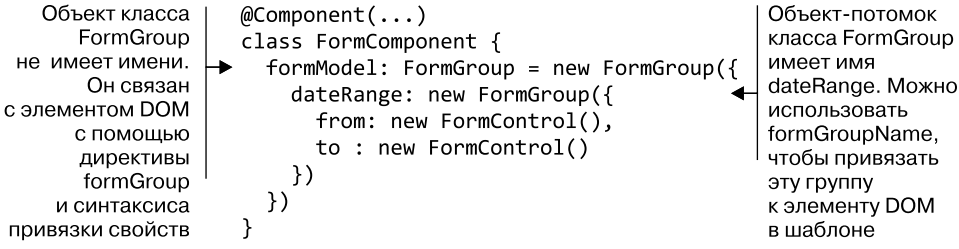
Затем добавьте атрибут `formGroup` в элемент HTML. Значение атрибута `formGroup` ссылается на свойство компонента, которое содержит экземпляр класса `FormGroup`:

```
<form [formGroup]="formModel"></form>
```

formGroupName

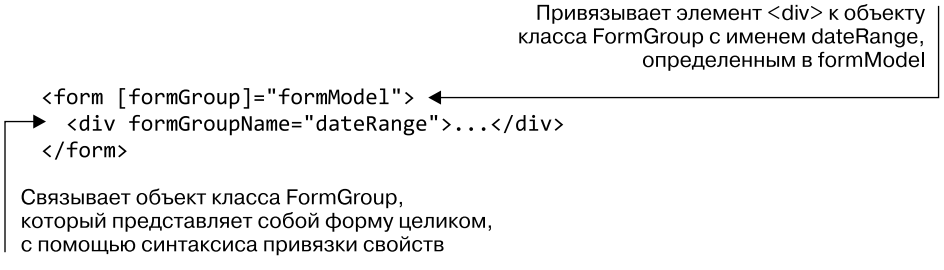
Может использоваться в целях привязки вложенных групп в форме. Вам нужно, чтобы эта директива находилась в области видимости директивы-предка `formGroup` для привязки одного из его экземпляров-потомков `FormGroup`. Модель формы, которую можно применять с `formGroupName`, определяется следующим образом (листинг 7.6).

Листинг 7.6. Модель формы, которую можно использовать с formGroupName



Теперь рассмотрим шаблон (листинг 7.7).

Листинг 7.7. Шаблон formGroup



В области видимости `formGroup` можно использовать `formGroupName`, чтобы связать классы-потомки модели с помощью имен, определенных в объекте-предке класса `FormGroup`. Значение, присваиваемое атрибуту `formGroupName`, должно совпадать с именем, выбранным для объекта-потомка класса `FormGroup` в листинге 7.7 (в данном случае это `dateRange`).

Сокращенный синтаксис привязки свойств

Поскольку значение, которое вы присваиваете директиве `*Name`, является строковым литералом, можете использовать сокращенный синтаксис и убрать квадратные скобки, окружающие имя атрибута. Полная версия будет выглядеть следующим образом:

```
<div [formGroupName]="'dateRange'">...</div>
```

Обратите внимание на квадратные скобки, окружающие имя атрибута, и одинарные кавычки, окружающие значение атрибута.

formControlName

Должна использоваться в области видимости директивы `formGroup`. Привязывает один из экземпляров-потомков класса `FormControl` к элементу DOM.

Продолжим пример с диапазоном дат, который мы начали разрабатывать, когда объясняли принцип действия директивы `formGroupName`. Компонент и модель формы остаются такими же. Вам нужно только завершить шаблон (листинг 7.8).

Листинг 7.8. Завершенный шаблон formGroup

```
<form [formGroup]="formModel">
  <div formGroupName="dateRange">
    <input type="date" formControlName="from">
    <input type="date" formControlName="to">
  </div>
</form>
```

Так же как и в случае использования директивы `formGroupName`, вы всего лишь указываете имя объекта класса `FormControl`, который хотите привязать к элементу DOM. Опять же эти имена вы выбрали при определении модели формы.

formControl

Можно использовать для форм с одним полем ввода в случае, когда вы не хотите создавать модель формы с помощью объекта класса `FormGroup`, но все еще хотите задействовать функции Forms API, такие как валидация и реактивное поведение, предоставленные свойством `FormControl.valueChanges`. Вы видели пример в главе 5, когда мы обсуждали наблюдаемые потоки. Вот суть того примера (листинг 7.9).

Листинг 7.9. Класс FormControl

Вместо того чтобы определять модель формы с помощью объекта класса FormGroup, как мы делали ранее в данном разделе, создайте отдельный экземпляр объекта класса FormControl

```
@Component({...})
class FormComponent {
  weatherControl: FormControl = new FormControl();

  constructor() {
    this.weatherControl.valueChanges
        .debounceTime(500)
        .switchMap(city => this.getWeather(city))
        .subscribe(weather => console.log(weather));
  }
}
```

Использует событие valueChanges для получения значения из формы

Вы можете применять директиву `ngModel`, чтобы синхронизировать значение, введенное пользователем, со свойством компонента; но, поскольку вы задействуете Forms API, можете использовать его реактивные функции. В предыдущем примере вы применили несколько операторов RxJS для наблюдаемого потока, возвращенного свойством `valueChanges`, для удобства пользователя. Более подробную информацию об этом примере вы найдете в главе 5.

Рассмотрим шаблон компонента `FormComponent` из листинга 7.9:

```
<input type="text" [formControl]="weatherControl">
```

Ввиду того что вы работаете с самостоятельным объектом класса `FormControl`, который не является частью группы `FormGroup`, вы не можете использовать директиву `formControlName` для привязки его по имени. Вместо этого применяете `formControl` и синтаксис привязки свойств.

formArrayName

Должна использоваться в области видимости директивы `formGroup`. Привязывает один из экземпляров-потомков класса `FormArray` к элементу DOM. Ввиду того что элементы управления формы объекта класса `FormArray` не имеют имен, вы можете привязать их к элементам DOM только по индексу. Обычно вы отрисовываете их в цикле с помощью директивы `ngFor`.

Рассмотрим пример, в котором пользователям разрешено вводить произвольное количество адресов электронной почты (листинг 7.10). Здесь мы выделим только ключевые части кода, но вы можете найти полный рабочий вариант примера в файле `02_growable-items-form.ts` в коде, поставляемом вместе с книгой. Сначала вы определяете модель.

Листинг 7.10. Файл 02_growable-items-form.ts: определение модели

```
@Component(...)
class AppComponent {
  formModel: FormGroup = new FormGroup({
    emails: new FormArray([
      new FormControl()
    ])
  });
  //...
}
```

Создает объект класса FormGroup, который будет представлять собой форму

Использует FormArray для создания коллекции адресов электронной почты, чтобы позволить пользователям вводить несколько адресов электронной почты

В шаблоне поля для ввода адресов электронной почты отрисованы в цикле с помощью директивы `ngFor` (листинг 7.11).

Листинг 7.11. Файл 02_growable-items-form.ts: шаблон

```
<ul formArrayName="emails">
  <li *ngFor="let e of formModel.get('emails').controls; let i=index">
    <input [formControlName]="i">
  </li>
  <button type="button" (click)="addEmail()">Add Email</button>
</ul>
```

Директива formArrayName привязывает объект класса FormArray к элементу DOM

Проходит в цикле по массиву адресов электронной почты и создает поле ввода для каждой записи

Привязывает элемент <input> к экземпляру объекта класса FormControl по индексу

Определяет обработчик события click

Запись `let i` в цикле `*ngFor` позволяет автоматически привязывать значение индекса массива к переменной `i`, доступной в цикле. Директива `formControlName` привязывает объект класса `FormControl` в классе `FormArray` к элементу DOM; но вместо того, чтобы указать его имя, она использует переменную `i`, которая ссылается на индекс текущего элемента управления. Когда пользователи нажимают

кнопку Add Email (Добавить адрес электронной почты), вы передаете новый экземпляр объекта класса `FormControl` в класс `FormArray`: `this.formModel.get('emails') .push(new FormControl())`.

На рис. 7.3 изображена форма с двумя полями ввода адресов электронной почты; анимированная версия, доступная на https://www.manning.com/books/angular-2-development-with-typescript, показывает принцип работы. Каждый раз, когда пользователь нажимает кнопку Add Email (Добавить адрес электронной почты), новый экземпляр объекта класса `FormControl` помещается в объект класса `FormArray`, который содержит адреса электронной почты, и через привязку данных новое поле ввода отрисовывается на странице. Кроме того, благодаря привязке данных значение формы внизу обновляется в реальном времени.

Рис. 7.3. Форма, содержащая растущую коллекцию адресов электронной почты

7.3.3. Переработка формы-примера

Переработаем код образца — регистрационной формы пользователя — из листинга 7.1. Изначально это была форма, написанная на простом HTML, а потом вы применили шаблон-ориентированный подход. Теперь сделаем реактивную версию. Начнем с определения модели формы (листинг 7.12).

Листинг 7.12. Определение модели формы

```
@Component(...)
class AppComponent {
  formModel: FormGroup;

  constructor() {
    this.formModel = new FormGroup({
      'username': new FormControl(),
      'ssn': new FormControl(),
      'passwordsGroup': new FormGroup({
        'password': new FormControl(),
        'pconfirm': new FormControl()
      })
    });
  }

  onSubmit() {
    console.log(this.formModel.value);
  }
}
```

Объявляет свойство компонента, которое содержит модель формы

Инициализирует модель формы в конструкторе

Вложенная группа для полей ввода паролей

Получает доступ к значению формы, используя свойство компонента formModel

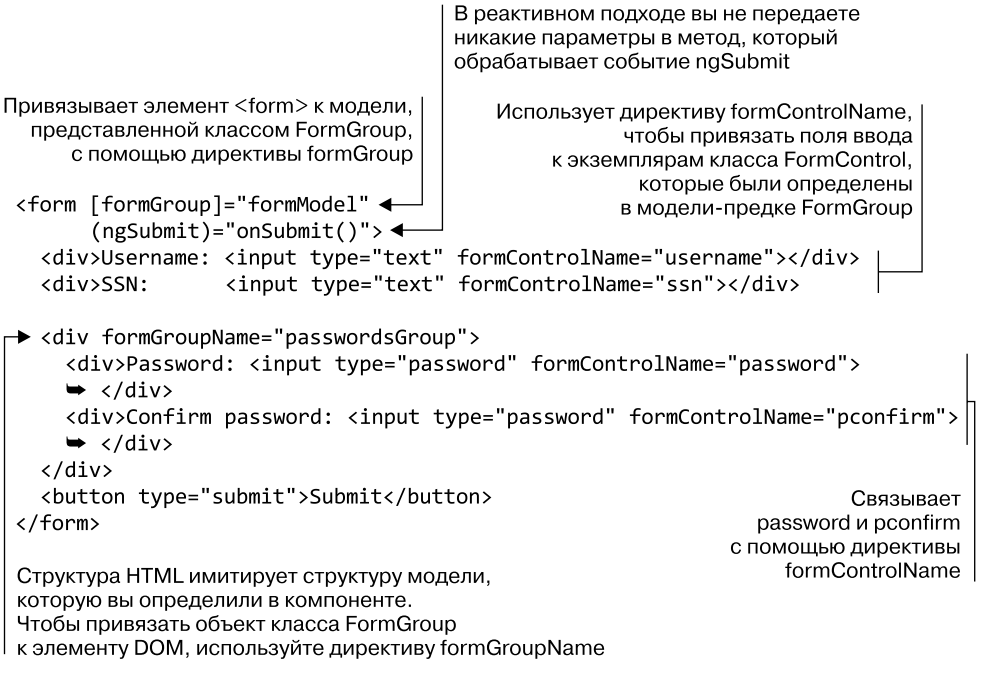
Свойство `formModel` содержит экземпляр класса `FormGroup`, определяющий структуру формы. Вы будете использовать это свойство в шаблоне, чтобы связать модель с элементом DOM с помощью директивы `formGroup`. Оно инициализируется программно в конструкторе путем создания экземпляров классов модели. Имена, которые вы даете элементам управления формы в объектах-предках класса `FormGroup`, применяются в шаблоне для привязки модели к элементу DOM с помощью директив `formControlName` и `formGroupName`.

Группа `passwordsGroup` является вложенным объектом класса `FormGroup`, который объединяет поля ввода и подтверждения паролей. Будет удобно управлять их значениями как одним объектом, когда вы добавите валидацию данных формы.

Ввиду того что директивы реактивных форм нельзя экспортировать, вы не можете получить к ним доступ в шаблоне и передать их напрямую в метод `onSubmit()` в качестве аргумента. Вместо этого вы получаете доступ к значению путем использования свойства компонента, которое хранит модель формы.

Теперь, когда мы определили модель, можете написать разметку HTML, привязываемую к модели (листинг 7.13).

Листинг 7.13. Привязка HTML к модели



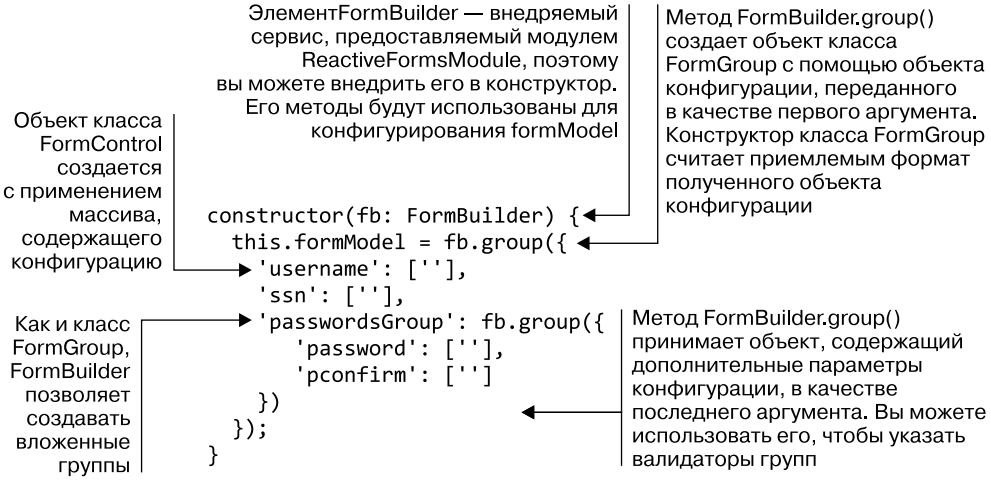
Поведение этой реактивной версии регистрационной формы идентично поведению шаблон-ориентированной формы, однако они отличаются внутренней реализацией. Полное приложение, иллюстрирующее, как создавать реактивные формы, находится в файле `03_reactive.ts` в коде, который поставляется вместе с книгой.

7.3.4. Использование FormBuilder

`FormBuilder` упрощает создание реактивных форм. Он не предоставляет никакой уникальной функциональности по сравнению с прямым использованием классов `FormControl`, `FormGroup` и `FormArray`, но его API более краткий, что избавляет от необходимости постоянно вводить имена классов.

Переработаем класс компонента из предыдущего раздела так, чтобы в нем применялся класс `FormBuilder`. Шаблон останется точно таким же, но вы измените способ создания `formModel`. Вот как он должен выглядеть (листинг 7.14).

Листинг 7.14. Переработка formModel с использованием класса FormBuilder



В отличие от класса `FormGroup` класс `FormBuilder` позволяет создавать объекты класса `FormControl` с помощью массива. Каждый элемент массива имеет особое значение. Первый элемент — изначальное значение объекта класса `FormControl`. Второй — функция-валидатор. Он также принимает третий аргумент, который является асинхронной функцией-валидатором. Остальными элементами массива можно пренебречь.

Как вы можете заметить, конфигурирование модели формы с помощью `FormBuilder` выглядит менее объемно и основывается на объектах конфигурации, а не на явном создании объектов классов элементов управления. Полное приложение, иллюстрирующее использование класса `FormBuilder`, находится в файле `04_form` в коде, который поставляется вместе с книгой.

7.4. Валидация данных формы

Одним из преимуществ использования Forms API, по сравнению с обычной привязкой данных, является тот факт, что вы можете провести валидацию данных формы. Валидация доступна для обоих типов форм: шаблон-ориентированной и реактивной. Вы создаете валидаторы как простые функции TypeScript. В реак-

тивном подходе вы применяете функции напрямую, в шаблон-ориентированном подходе оборачиваете их в пользовательские директивы.

Начнем с валидации реактивных форм, а потом перейдем к шаблон-ориентированным. Мы рассмотрим основы и применим валидацию к нашему примеру — регистрационной форме.

7.4.1. Валидация реактивных форм

Валидаторы — обычные функции, которые имеют следующий интерфейс:

```
interface ValidatorFn {
  (c: AbstractControl): {[key: string]: any};
}
```

Объявление типа {[key: string]: any} описывает объектный литерал, где имена свойств являются строками и значения могут быть любого типа

Функция валидатора должна объявлять один параметр типа `AbstractControl` и возвращать объектный литерал. Реализация функции не ограничена — все зависит только от автора валидатора. Параметр `AbstractControl` является суперклассом для классов `FormControl`, `FormGroup` и `FormArray`, и это значит, что валидаторы могут быть созданы для всех классов модели.

Несколько заранее определенных валидаторов поставляется с Angular: `required`, `minLength`, `maxLength` и `pattern`. Они определяются как статические методы класса `Validators`, объявленного в модуле `@angular/forms`, и соответствуют стандартным атрибутам валидации HTML5.

Как только у вас появится валидатор, нужно будет настроить модель для его использования. В реактивном подходе вы предоставляете валидаторы в качестве аргументов для конструкторов классов модели. Рассмотрим пример:

Первый параметр является значением по умолчанию, а второй — функцией валидатора

```
import { FormControl, Validators } from '@angular/forms';

let usernameControl = new FormControl('', Validators.required);
```

Вы также можете предоставить список валидаторов в качестве второго аргумента:

```
let usernameControl = new FormControl('', [Validators.required,
➥ Validators.minLength(5)]);
```

Чтобы проверить корректность данных элемента управления, используйте свойство `valid`, которое возвращает либо `true`, либо `false`:

Указывает, проходит ли значение, введенное в поле, правила проверки, настроенные для элемента управления

```
let isValid: boolean = usernameControl.valid;
```

Если какое-либо правило проверки не пройдено, то вы можете получить объекты ошибок, сгенерированные функциями валидатора.

```
let errors: {[key: string]: any} = usernameControl.errors;
```

Объект ошибки

Ошибка, возвращенная валидатором, представлена объектом JavaScript, который имеет свойство с таким же именем, как и у валидатора. Является ли он объектным литералом или объектом со сложной прототипной цепочкой, для валидатора не имеет значения.

Значение свойства может иметь любой тип и предоставлять дополнительную информацию об ошибке. Например, стандартный валидатор `Validators.minLength()` возвращает следующий объект ошибки:

```
{
  minlength: {
    requiredLength: 7,
    actualLength: 5
  }
}
```

Объект имеет свойство высшего уровня, имя которого соответствует имени валидатора: `minlength`. Его значение также является объектом с двумя полями: `requiredLength` и `actualLength`. Эти детали могут использоваться, чтобы отображать удобное для пользователя сообщение об ошибке.

Не все валидаторы предоставляют дополнительную информацию об ошибке. Иногда свойство высшего уровня лишь указывает, что произошла ошибка. В этом случае свойство инициализируется со значением `true`. Ниже приведен пример стандартного объекта ошибки `Validators.required()`:

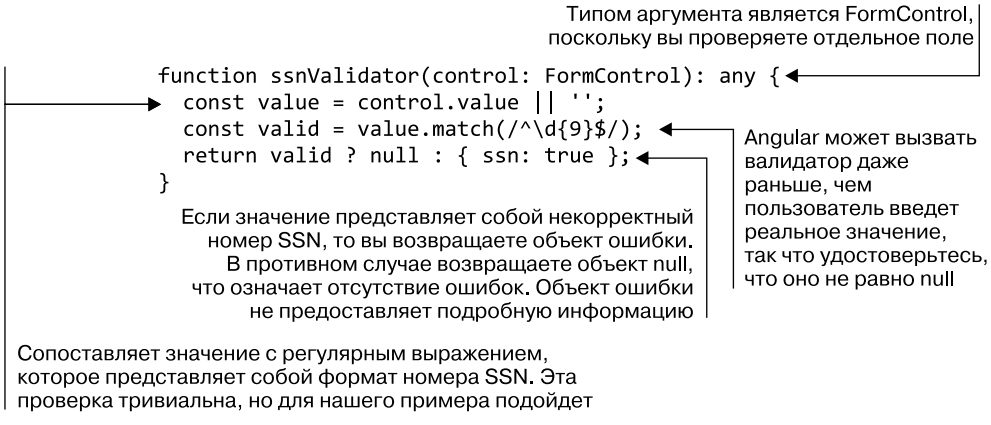
```
{
  required: true
}
```

Пользовательские валидаторы

Стандартные валидаторы хороши для валидации основных типов данных, таких как строки и числа. Если необходимо проверять более сложные типы данных или логику приложения, может понадобиться создать пользовательский валидатор. Ввиду того что валидаторы в Angular — всего лишь функции с определенной сигнатурой, их довольно просто создать. Нужно объявить функцию, принимающую экземпляр одного из типов элементов управления — `FormControl`, `FormGroup` или `FormArray` — и возвращающую объект, который представляет собой ошибку валидации (см. вставку «Объект ошибки»).

Ниже следует пример пользовательского валидатора, проверяющего, является ли значение, введенное в элементе управления, корректным номером социально-

го страхования (Social Security Number, SSN), представляющим собой уникальный идентификатор для каждого гражданина Соединенных Штатов:



Пользовательские валидаторы используются таким же образом, как и стандартные:

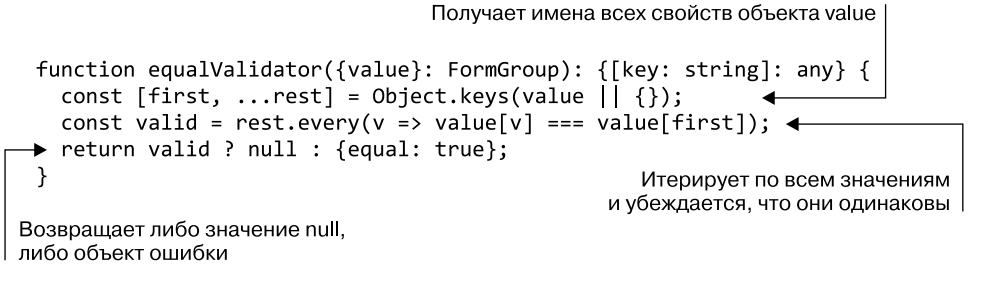
```
let ssnControl = new FormControl('', ssnValidator);
```

Полное работающее приложение, иллюстрирующее, как создавать пользовательские валидаторы, находится в файле `05_custom-validator.ts` в коде, который поставляется вместе с книгой.

Валидаторы для групп

Вам может понадобиться проверять значения не только отдельных полей, но и их групп. Angular также позволяет определять функции-валидаторы для объектов класса `FormGroup`.

Создадим валидатор, который гарантирует, что поля `password` и `password-confirmation` из нашего примера формы регистрации имеют одинаковые значения. Одна из возможных реализаций выглядит следующим образом:



Сигнатура функции соответствует интерфейсу `ValidatorFn`: первый параметр имеет тип `FormGroup`, который является подклассом `AbstractControl`, а ее возвращаемый тип — объектный литерал. Обратите внимание на то, что вы используете свойство ECMAScript, называемое *деструктурированием* (см. одноименный

подраздел в приложении А). Вы извлекаете свойство `value` из экземпляра класса `FormGroup`, которое будет передано в качестве аргумента. Здесь это имеет смысл, поскольку вы никогда не получаете доступ к какому-либо другому свойству объекта класса `FormGroup` в коде валидатора.

Далее вы получаете имена всех свойств в объекте `value` и сохраняете их в двух переменных, `first` и `rest`. Переменная `first` — это имя свойства, которое будет использовано в качестве ссылочного значения; значения всех других свойств должны быть одинаковыми, чтобы пройти проверку. В переменной `rest` хранятся имена всех других свойств. Вы снова применяете функциональность деструктурирования для извлечения ссылок на элементы массива (см. подраздел А.5.3 «Деструктурирование» в приложении А). Наконец, вы возвращаете либо значение `null`, если значения группы являются допустимыми, либо в противном случае объект, который указывает на ошибку.

Выполнение валидации для примера формы регистрации

Теперь, когда мы рассмотрели основы, добавим валидацию в наш пример. Вы будете использовать валидаторы `ssnValidator` и `equalValidator`, которые мы реализовали ранее в этом разделе. Далее представлена модифицированная модель формы (листинг 7.15).

Листинг 7.15. Модифицированная модель формы

```
this.formModel = new FormGroup({
  'username': new FormControl('', Validators.required),
  'ssn': new FormControl('', ssnValidator),
  'passwordsGroup': new FormGroup({
    'password': new FormControl('', Validators.minLength(5)),
    'pconfirm': new FormControl('')
  }, {}, equalValidator)
});
```

Для элемента управления username вы используете стандартный валидатор Validators.required. Он позволяет гарантировать, что поле не оставлено пустым

Для поля ssn применяете валидатор ssnValidator, реализованный ранее

Для поля password задействуете стандартный валидатор validators.minLength, который возвращает ошибку в случае, если введенное строковое значение содержит меньше пяти символов

Конфигурирует валидатор equalValidator для группы passwordsGroup. Он гарантирует, что все поля в группе имеют одинаковое значение. В отличие от класса FormControl вы передаете валидаторы в качестве третьего аргумента в конструкторе класса FormGroup

Прежде чем выводить модель формы в консоли с помощью метода `onSubmit()`, вы проверяете, является ли форма действительной:

```
onSubmit() {
  if (this.formModel.valid) {
    console.log(this.formModel.value);
  }
}
```

В подходе, ориентированном на работу с моделями, для конфигурирования валидаторов требуется внести изменения только в код, но вам также нужно сделать несколько изменений в шаблоне. Следует отобразить ошибки валидации, когда пользователь вводит недопустимое значение. Рассмотрим модифицированную версию шаблона (листинг 7.16).

Листинг 7.16. Измененный шаблон

```
<form [formGroup]="formModel" (ngSubmit)="onSubmit()" novalidate>
  <div>Username:
    <input type="text" formControlName="username">
    <span [hidden]="!formModel.hasError('required', 'username')">Username is
    ➥ required</span>
  </div>
  <div>SSN:
    <input type="text" formControlName="ssn">
    <span [hidden]="!formModel.hasError('ssn', 'ssn')">SSN is invalid
    ➥ </span>
  </div>

  <div formGroupName="passwordsGroup">
    <div>Password:
      <input type="password" formControlName="password">
      <span [hidden]="!formModel.hasError('minlength', ['passwordsGroup',
      ➥ 'password'])">
          Password is too short
      </span>
    </div>
    <div>Confirm password:
      <input type="password" formControlName="pconfirm">
      <span [hidden]="!formModel.hasError('equal', 'passwordsGroup')">
          Passwords must be the same
      </span>
    </div>
  </div>
  <button type="submit">Submit</button>
</form>
```

При определенных условиях показывает сообщение об ошибке для поля username

Добавляет сообщение об ошибке для поля ssn, как вы это сделали для поля username

При определенных условиях показывает сообщение об ошибке для поля password

Обратите внимание на то, как вы получаете доступ к методу `hasError()`, доступному в модели формы, когда при определенных условиях показываете сообщения об ошибке. Он принимает два параметра: имя ошибки валидации, наличие которой вы хотите проверить, и путь к необходимому полю в модели формы. В случае поля `username` оно является прямым потомком объекта класса `FormGroup` высшего уровня, который представляет собой модель формы, так что вы просто указываете имя элемента управления. Но поле `password` — потомок вложенного объекта класса `FormGroup`, поэтому путь к элементу управления указывается как массив строк. Первый элемент — имя вложенной группы, а второй — собственно имя поля `password`. Как и поле `username`, группа `passwordsGroup` указывает путь как строку ввиду того, что она является прямым потомком объекта высшего уровня класса `FormGroup`.

Полное работающее приложение, иллюстрирующее, как использовать функции валидатора с реактивными формами, находится в файле `09_reactive-with-validation.ts` в коде, поставляемом вместе с книгой. В этом примере вы жестко

закодировали сообщения об ошибках в шаблоне, но их можно предоставить, применяя валидаторы. Чтобы увидеть пример, в котором сообщения об ошибках предоставляются динамически, см. файл `07_custom-validator-error-message.ts`.

Конфигурирование валидаторов с помощью FormBuilder

Валидаторы также можно сконфигурировать, когда вы задействуете `FormBuilder` для определения моделей форм. Ниже приведена модифицированная версия модели нашего примера — регистрационной формы — с использованием `FormBuilder`:

```
@Component(...)
class AppComponent {
  formModel: FormGroup;

  constructor(fb: FormBuilder) {
    this.formModel = fb.group({
      'username': ['', Validators.required],
      'ssn': ['', ssnValidator],
      'passwordsGroup': fb.group({
        'password': ['', Validators.minLength(5)],
        'pconfirm': ['']
      }, {validator: equalValidator})
    });
  }
}
```

FormBuilder — зарегистрированный поставщик, поэтому вы можете внедрять его в конструктор компонента вместо того, чтобы создавать его объект непосредственно с помощью ключевого слова new

При конфигурировании FormControls валидаторы указываются как второй элемент

При конфигурировании валидаторов для FormGroup вы предоставляете объект options в качестве второго аргумента метода group(), а также используете свойство объекта validator для указания валидаторов

Асинхронные валидаторы

Forms API поддерживает асинхронные валидаторы. Они могут использоваться для проверки значений форм с помощью удаленного сервера, что включает в себя отправку HTTP-запроса. Как и обычные, асинхронные валидаторы являются функциями. Единственное отличие состоит в том, что асинхронные валидаторы должны возвращать объекты типов `Observable` или `Promise`. Ниже приведен асинхронный вариант валидатора для номера SSN (листинг 7.17).

Листинг 7.17. Асинхронный валидатор для номера SSN

```
function asyncSsnValidator(control: FormControl): Observable<any> {
  const value: string = control.value || '';
  const valid = value.match(/^\d{9}$/);
  return Observable.of(valid ? null : { ssn: true }).delay(5000);
}
```

В данном случае возвращаемое значение будет иметь тип Observable

Чтобы этот пример оставался простым, вы эмулируете асинхронность с помощью оператора задержки из библиотеки RxJS. Результат валидации поступает через пять секунд после вызова функции

Асинхронные валидаторы передаются в качестве третьего аргумента в конструкторы классов модели:

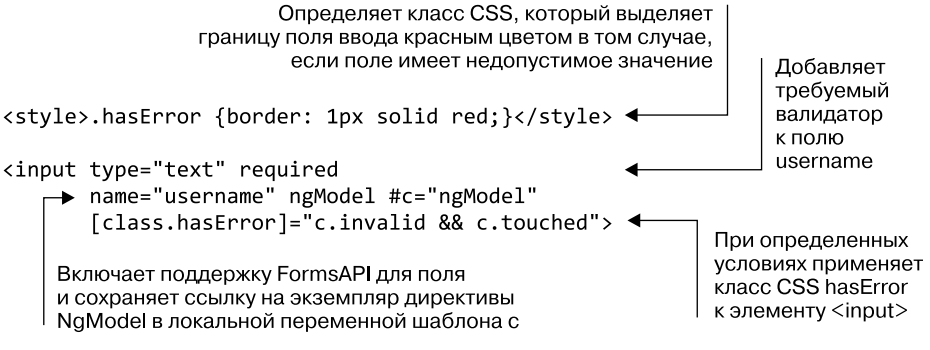
```
let ssnControl = new FormControl('', null, asyncSsnValidator);
```

Полное работающее приложение, иллюстрирующее, как использовать асинхронные валидаторы, находится в файле `08_async-validator.ts` в коде, который поставляется вместе с книгой.

Проверка состояния и допустимости значения поля

Вы уже знакомы с такими свойствами элементов управления, как `valid`, `invalid` и `errors`, используемыми для проверки состояния поля. Здесь мы рассмотрим некоторые другие свойства, предназначенные для повышения удобства применения.

- *Затронутые и незатронутые поля*. В дополнение к проверке допустимости значений элементов управления вы также можете использовать свойства `touched` и `untouched`, чтобы проверить, работал ли пользователь с данным полем. Если пользователь помещает поле в фокус с помощью клавиатуры или мыши, то оно считается затронутым; в противном случае — незатронутым. Это может быть полезно при отображении сообщений об ошибках: если поле имеет недопустимое значение, но пользователь с ним не работал, то вы можете выбрать не выделять его красным цветом, поскольку это не ошибка пользователя. Рассмотрим следующий пример:



ПРИМЕЧАНИЕ

Все свойства, рассмотренные в этом примере, доступны в классах модели FormControl, FormGroup и FormArray, а также в шаблон-ориентированных директивах NgModel, NgModelGroup и NgForm.

Обратите внимание на пример привязки класса CSS в последней строке. При определенных условиях класс CSS `hasError` применяется для элемента, если выражение справа имеет значение `true`. При использовании только свойства `c.invalid` граница поля будет выделена в процессе отрисовки страницы; но такое действие может смутить пользователей, особенно если на странице много полей ввода. Вместо этого вы добавляете еще одно условие: поле должно быть затронутым. Теперь оно выделяется только после того, как пользователь поработает с ним.

- *Чистые и грязные поля*. Еще одна пара полезных свойств: `pristine` и его противоположность `dirty`. Свойство `dirty` указывает на то, что поле было изменено после инициализации. Эти свойства могут быть использованы для напоминания пользователю о необходимости сохранить данные, прежде чем он покинет страницу или закроет диалоговое окно.

ПРИМЕЧАНИЕ

Все вышеупомянутые свойства имеют соответствующие классы CSS (ng-touched и ng-untouched, ng-dirty и ng-pristine, ng-valid и ng-invalid), которые автоматически добавляются к элементам HTML, когда свойство имеет значение true. Это может быть полезно, если нужно задать отдельный стиль для элементов, находящихся в особом состоянии.

- *Ожидающие поля*. Если для элемента управления определены асинхронные валидаторы, вам также может пригодиться булево свойство `pending`. Оно показывает, что допустимость значения в данный момент не определена. Это случается, когда асинхронный валидатор еще работает и нужно ждать результат. Данное свойство можно использовать для отображения индикатора прогресса.

Для реактивных форм свойство `statusChanges` типа `Observable` может быть более удобным. Оно отправляет одно из трех значений: `VALID`, `INVALID` и `PENDING`.

Валидация шаблон-ориентированных форм

Директивы — все, что можно использовать при создании шаблон-ориентированных форм, поэтому можно обернуть функции-валидаторы в директивы, чтобы применять их в шаблоне. Создадим директиву, которая оборачивает реализованный в листинге 7.17 валидатор SSN (листинг 7.18).

Листинг 7.18. Директива SsnValidatorDirective

```
@Directive({
  selector: '[ssn]',
  providers: [{
    provide: NG_VALIDATORS,
    useValue: ssnValidator,
    multi: true
  }]
})
class SsnValidatorDirective {}
```

Определяет селектор директивы как атрибут HTML

Объявляет директиву с помощью аннотации @Directive из модуля @angular/core

Регистрирует валидатор ssnValidator в качестве поставщика NG_VALIDATORS

Квадратные скобки, окружающие селектор `ssn`, указывают на то, что директива может быть использована в качестве атрибута. Это удобно, поскольку можно добавить атрибут к любому элементу `<input>` или к компоненту Angular, представленному как пользовательский элемент HTML.

В этом примере вы регистрируете функцию-валидатор с помощью заранее определенного токена Angular `NG_VALIDATORS`. Данный токен, в свою очередь, внедряется директивой `NgModel`, и та получает список всех валидаторов, прикрепленных к элементу HTML.

Далее NgModel передает экземпляру класса FormControl валидаторы, которые создаются ею неявно. Этот же механизм отвечает за запуск валидаторов; директивы — всего лишь альтернативный способ их конфигурирования. Свойство multi позволяет связывать множество значений с одним токеном. Когда токен внедрен в директиву NgModel, она получает список значений вместо одного. Это позволяет передавать несколько валидаторов. Вот как можно использовать директиву SsnValidatorDirective:

```
<input type="text" name="my-ssn" ngModel ssn>
```

Вы можете найти полное работающее приложение, в котором демонстрируется использование валидаторов-директив, в файле 06_custom-validator-directive.ts в коде, поставляемом вместе с книгой.

7.4.2. Выполнение валидации для примера формы регистрации

Теперь вы можете добавить валидацию формы для нашего примера. Начнем с шаблона (листинг 7.19).

Листинг 7.19. Шаблон валидации для формы регистрации

Добавляет директиву валидации в качестве обязательного атрибута

Передает значение формы и состояние корректности данных в метод onSubmit()

```
<form #f="ngForm" (ngSubmit)="onSubmit(f.value, f.valid)" novalidate>
  <div>Username:
    <input type="text" name="username" ngModel required>
    <span [hidden]="!f.form.hasError('required', 'username')">Username is
    ➥ required</span>
  </div>
  <div>SSN:
    <input type="text" name="ssn" ngModel ssn>
    <span [hidden]="!f.form.hasError('ssn', 'ssn')">SSN in invalid</span>
  </div>

  <div ngModelGroup="passwordsGroup" equal>
    <div>Password:
      <input type="password" name="password" ngModel minlength="5">
      <span [hidden]="!f.form.hasError('minlength', ['passwordsGroup',
      ➥ 'password'])">
        Password is too short
      </span>
    </div>
    <div>Confirm password:
      <input type="password" name="pconfirm" ngModel>
      <span [hidden]="!f.form.hasError('equal', 'passwordsGroup')">
        Passwords must be the same
      </span>
    </div>
  </div>
  <button type="submit">Submit</button>
</form>
```

При определенных условиях показывает и скрывает сообщения об ошибках

Equal — это директива-оболочка для валидатора equalValidator, который вы реализовали ранее. Пользовательские валидаторы-директивы добавляются точно таким же образом, как и стандартные

При шаблон-ориентированном подходе у вас нет модели в компоненте. Только шаблон может проинформировать обработчик формы о том, что она имеет допустимые значения, и поэтому вы передаете значение формы и состояние ее корректности как аргументы в метод `onSumbit()`. Вы также добавляете атрибут `novalidate`, чтобы браузер не выполнял собственную валидацию, мешая Angular совершать свою.

Директивы-валидаторы добавляются как атрибуты. Необходимая директива предоставляется Angular и становится доступна, когда вы зарегистрируете поддержку Forms API для объекта класса `FormModule`. Аналогично вы можете использовать директиву `minlength` для валидации поля `password`.

Чтобы ситуативно показывать и скрывать сообщения об ошибках валидации, вы используете тот же метод `hasError()`, применяемый в реактивной версии. Но для получения доступа к этому методу нужно задействовать свойство формы типа `FormGorup`, доступное в переменной `f`, которая ссылается на экземпляр директивы `formGroup`.

В методе `onSubmit()` вы проверяете корректность формы до того, как выведете значение в консоли (листинг 7.20).

Листинг 7.20. Проверка валидации формы

```
@Component({ template: '...' })
class AppComponent {
  onSubmit(formValue: any, isFormValid: boolean) {
    if (isFormValid) {
      console.log(formValue);
    }
  }
}
```

Добавляет параметр isFormValid в определение метода

Выводит значение формы в консоли в случае, если форма корректна

Остался последний шаг: нужно добавить пользовательский валидатор в список объявлений директивы `NgModule`, где вы определили компонент `AppComponent` (листинг 7.21).

Листинг 7.21. Добавление валидаторов-директив

```
@NgModule({
  imports :      [ BrowserModule, FormsModule ],
  declarations:  [ AppComponent, EqualValidatorDirective,
  ➥ SsnValidatorDirective ],
  bootstrap :    [ AppComponent ]
  })
class AppModule {}
```

Добавляет директивы в список объявлений

Полное работающее приложение, иллюстрирующее, как использовать валидаторы-директивы для шаблон-ориентированных форм, находится в файле `10_template-driven-with-validation.ts` в коде, который поставляется вместе с книгой.

7.5. Практикум: добавление валидации в форму поиска

Это упражнение начнется с того места, где мы остановились в главе 6. Вам нужно изменить код компонента `SearchComponent` так, чтобы включить валидацию формы и собрать данные, введенные в нее.

Когда данные с формы отправляются, вы выводите на экран значение формы в консоли браузера. Глава 8 посвящена общению с сервером, в ней вы переработаете код так, чтобы форма поиска выполняла реальный HTTP-запрос.

В этом разделе вы сделаете следующие шаги.

1. Добавите новый метод в класс `ProductService`, который возвращает массив всех доступных категорий продуктов.
2. Создадите модель, представляющую форму поиска, с помощью `FormBuilder`.
3. Сконфигурируете правила валидации для модели.
4. Переработаете шаблон так, чтобы в нем выполнялась привязка свойств для модели, созданной на предыдущем шаге.
5. Реализуете метод `onSearch()` для обработки события формы `submit`.

На рис. 7.4 показано, как будет выглядеть форма поиска после завершения упражнения. Отображены валидаторы в действии.

Если вы хотели бы увидеть финальную версию этого проекта, то откройте исходный код в каталоге `auction` для главы 7. В противном случае скопируйте каталог `auction` в другое место и следуйте инструкциям из данного раздела.

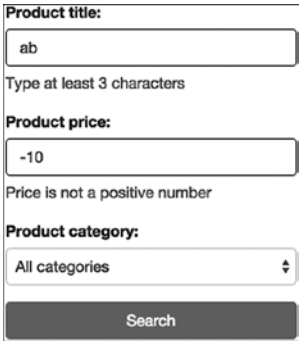


Рис. 7.4. Форма поиска с валидаторами

7.5.1. Изменение корневого модуля для добавления поддержки Forms API

Обновите файл `app.module.ts`, чтобы включить поддержку реактивных форм для вашего приложения (листинг 7.22). Импортируйте модуль `ReactiveFormsModule` из `@angular/forms` и добавьте его в список импортированных модулей в основном приложении `NgModule`.

Листинг 7.22. Обновленный файл app.module.ts

```
import { ReactiveFormsModule } from '@angular/forms';
@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    ReactiveFormsModule,
    RouterModule.forRoot([ ... ])
  ],
```

7.5.2. Добавление списка категорий в компонент SearchComponent

Каждый продукт имеет свойство `categories`, представленное массивом строк; один продукт может относиться к нескольким категориям. Форма должна позволять пользователю выбирать категорию во время поиска продуктов; нужен способ предоставить список всех доступных категорий форме, чтобы она могла отобразить их для пользователей.

В реальных приложениях категории, скорее всего, будут приходить с сервера. В этом примере онлайн-аукциона вы добавите метод в класс `ProductService`, который вернет жестко закодированные категории.

1. Откройте файл `app/services/product-service.ts` и добавьте метод `getAllCategories()`, который не принимает никаких параметров и возвращает список строк:

```
getAllCategories(): string[] {
  return ['Books', 'Electronics', 'Hardware'];
}
```

2. Откройте файл `app/components/search/search.ts` и добавьте оператор импорта для `ProductService`:

```
import {ProductService} from '../../services/product-service';
```

3. Сконфигурируйте этот сервис как поставщик для компонента `SearchComponent`:

```
@Component({
  selector: 'auction-search',
  providers: [ProductService],
  //...
})
```

4. Объявите свойство класса `categories: string[]` как ссылку на список категорий. Вы будете использовать ее для привязки данных:

```
export default class SearchComponent {
  categories: string[];
}
```

5. Объявите метод `constructor` с одним параметром: `ProductService`. Angular внедрит его при создании объекта компонента. Инициализируйте свойство `categories`, используя метод `getAllCategories()`:

Ключевое слово private автоматически создает свойство класса с таким же именем, как у параметра, и инициализирует его с помощью предоставленного значения

```
constructor(private productService: ProductService) {
  this.categories = this.productService.getAllCategories();
}
```

7.5.3. Создание модели формы

Теперь определим модель, которая будет обрабатывать форму поиска.

1. Откройте файл `app/components/search/search.ts` и добавьте оператор импорта для Forms API. Оператор `import` в начале файла должен выглядеть так:

```
import {Component} from '@angular/core';
import {FormControl, FormGroup, FormBuilder, Validators} from
➥ '@angular/forms';
```

2. Объявите свойство класса `formModel` для типа `FormGroup`:

```
export default class SearchComponent {
  formModel: FormGroup;
  //...
}
```

3. В конструкторе определите `formModel` с помощью класса `FormBuilder`:

```
const fb = new FormBuilder();
this.formModel = fb.group({
  'title': [null, Validators.minLength(3)],
  'price': [null, positiveNumberValidator],
  'category': [-1]
})
```

4. Добавьте функцию `positiveNumberValidator`:

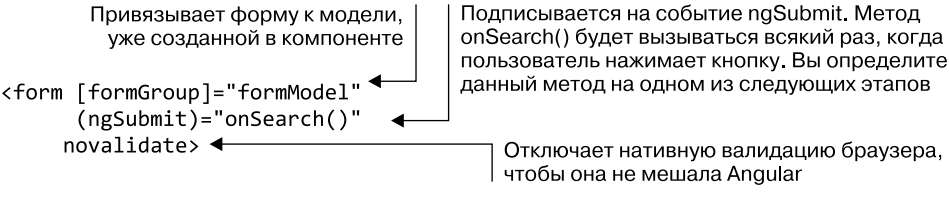
```
function positiveNumberValidator(control: FormControl): any {
  if (!control.value) return null;
  const price = parseInt(control.value);
  return price === null || typeof price === 'number' && price > 0
      ? null : {positivenumber: true};
}
```

Функция `positiveNumberValidator()` пробует проанализировать целочисленное значение, полученное от `FormControl`, с помощью стандартной функции `parseInt()`. Если проанализированное значение представляет собой положительное целое число, то функция вернет значение `null`, что означает отсутствие ошибок. В противном случае она вернет объект ошибки.

7.5.4. Переработка шаблона

Добавим в шаблон директивы форм, чтобы привязать модель, определенную в предыдущем шаге, к элементам HTML.

1. Вы определили модель формы с помощью реактивного подхода, поэтому в шаблоне должны прикрепить директиву `NgFormModel` к элементу `<form>`.



2. Определите правила валидации и при определенных условиях отображайте сообщения об ошибках для поля `title`:

```
<div class="form-group"
     [class.has-error]="formModel.hasError('minlength', 'title')">
  <label for="title">Product title:</label>
  <input id="title"
         placeholder="Title"
         class="form-control"
         type="text"
         formControlName="title"
         minlength="3">
  <span class="help-block"
        [class.hidden]="!formModel.hasError('minlength', 'title')">
    Type at least 3 characters
  </span>
</div>
```

Здесь вы используете классы CSS `form-group`, `form-control`, `has-error` и `help-block`, определенные в библиотеке Twitter Bootstrap. Они необходимы для того, чтобы соответствующим образом отрисовать форму и выделить поле красной границей в случае ошибки валидации. Более подробную информацию об этих классах вы можете найти в документации к Bootstrap в разделе Forms: https://getbootstrap.com/docs/3.3/css/#forms.

3. Сделайте то же самое для поля с ценой продукта:

```
<div class="form-group"
      [class.has-error]="formModel.hasError('positivenumber', 'price')">
  <label for="price">Product price:</label>
  <input id="price"
         placeholder="Price"
         class="form-control"
         type="number"
         step="any"
         min="0"
         formControlName="price">
  <span class="help-block"
        [class.hidden]="!formModel.hasError('positivenumber', 'price')">
    Price is not a positive number
  </span>
</div>
```

4. Добавьте правила валидации и сообщение об ошибке для поля категории продукта:

```
<div class="form-group">
  <label for="category">Product category:</label>
  <select id="category"
          class="form-control"
          formControlName="category">
    <option value="-1">All categories</option>
    <option *ngFor="let c of categories"
            [value]="c">{{c}}</option>
  </select>
</div>
```

Кнопка Submit (Отправить) не изменяется.

7.5.5. Реализация метода onSearch()

Добавьте следующий метод `onSearch()`:

```
onSearch() {
  if (this.formModel.valid) {
    console.log(this.formModel.value);
  }
}
```

7.5.6. Запуск онлайн-аукциона

Чтобы запустить приложение, откройте командную строку и запустите HTTP-сервер в корневом каталоге проекта. Введите http://localhost:8080 в браузере — вы должны увидеть главную страницу, которая содержит форму поиска, показанную на рис. 7.4. В этой версии приложения показывается создание формы и валидация без выполнения поиска. Вы реализуете функциональность поиска в главе 8, когда мы будем рассматривать способы общения с сервером.

7.6. Резюме

Из этой главы вы узнали, как работать с формами в Angular. Вот основные выводы.

- Существует два подхода к работе с формами: шаблон-ориентированный и реактивный. Первый проще, и его легче конфигурировать, но второй легче тестировать, он более гибок и предоставляет возможность управления формой.
- Реактивный подход более полезен для приложений, которые применяют не только отрисовщик DOM, но и какой-нибудь другой (например, NativeScript), нацеливаясь на небраузерные среды. Реактивные формы достаточно написать всего раз, после чего их можно использовать повторно для более чем одного отрисовщика.
- Какое-то количество стандартных валидаторов поставляется с Angular, но можно создавать собственные. Вы должны выполнять валидацию данных, введенных пользователем, но валидация на стороне клиента не может заменить валидацию на сервере. Считайте валидацию на стороне клиента способом предоставить пользователю мгновенную обратную связь с минимизацией количества запросов на сервер, содержащих недопустимые данные.

8 Взаимодействие с серверами с помощью HTTP и WebSockets

В этой главе:

- создание простого веб-сервера с использованием таких сред, как Node и Express;
- создание запросов к серверу из Angular с применением API `Http`-объектов;
- обмен данными с Node-сервером из Angular-клиентов с помощью протокола HTTP;
- заключение WebSocket-клиента в сервис Angular, генерирующий доступный для наблюдения поток;
- передача данных с сервера нескольким клиентам через WebSocket-объекты.

Angular-приложения могут обмениваться данными с любым веб-сервером, поддерживающим HTTP- или WebSocet-протоколы, независимо от того, какая из платформ используется на серверной стороне. До сих пор внимание уделялось в основном серверной стороне Angular-приложений, за исключением описанного в главе 5 примера с сервисом погоды. В настоящей главе вопрос обмена данными с веб-серверами будет рассмотрен более подробно.

Сначала мы представим краткий обзор принадлежащего Angular `Http`-объекта, а затем предложим создать веб-сервер с помощью TypeScript и Node.js. Этот сервер будет предоставлять данные для всех примеров кода, включая онлайн-аукцион. Затем мы рассмотрим вопросы создания в клиентском коде HTTP-запросов к веб-серверам и *наблюдаемых объектов*, описанных в главе 5. Кроме того, уделим внимание вопросу обмена данными с сервером через WebSocket-объекты, сделав акцент на принудительной доставке данных на серверную сторону.

В разделе «Практикум» вы реализуете функцию поиска товара, в которой данные по аукционным товарам и обзоры будут поступать с сервера через HTTP-запросы. Вдобавок вы реализуете уведомление о предлагаемой цене товара, отправляемое сервером с использованием WebSocket-протокола.

8.1. Краткий обзор API Http-объектов

В веб-приложениях HTTP-запросы запускаются в асинхронном режиме, сохраняющем отзывчивость пользовательского интерфейса и позволяющем пользователю продолжать работать с приложением в ходе обработки сервером этих запросов. Асинхронные HTTP-запросы могут реализовываться с использованием функций обратного вызова, промисов или наблюдаемых объектов. Хотя промисы и исключают все неудобства, связанные с функциями обратных вызовов (см. приложение А), они имеют следующие недостатки:

- отсутствие способов прекращения отложенного запроса, совершенного с помощью промиса;
- отсутствие со стороны промисов предложенного способа обработки продолжительного потока, состоящего из фрагментов данных, поступающих со временем. При разрешении промиса или его отклонении клиент получает либо данные, либо сообщение об ошибке, но в любом случае это будет неделимый фрагмент данных.

У наблюдателей такие недостатки отсутствуют. В подразделе 5.2.2 рассматривался сценарий на основе применения промиса, в результате чего для извлечения коммерческого предложения на акцию получалась масса ненужных запросов, порождающих ненужный сетевой трафик. Затем в подразделе 5.2.3, в примере с сервисом погоды, был показан способ прекращения HTTP-запросов, осуществляемый с помощью наблюдателей.

А теперь рассмотрим имеющуюся в Angular реализацию класса `Http`, включенного в пакет `@angular/http`. В него входят несколько классов и интерфейсов, как описано в документации по Angular HTTP-клиенту, которую можно найти на https://angular.io/guide/http. Если заглянуть в содержимое файла определения типа `@angular/http/src/http.d.ts`, то в классе `Http` можно увидеть следующие API:

```
import {Observable} from 'rxjs/Observable';
...
export declare class Http {
...
    constructor(_backend: ConnectionBackend, _defaultOptions: RequestOptions);
    request(url: string | Request, options?: RequestOptionsArgs):
    ➥ Observable<Response>;
    get(url: string, options?: RequestOptionsArgs): Observable<Response>;
    post(url: string, body: string, options?: RequestOptionsArgs):
    ➥ Observable<Response>;
    put(url: string, body: string, options?: RequestOptionsArgs):
    ➥ Observable<Response>;
  delete(url: string, options?: RequestOptionsArgs): Observable<Response>;
  patch(url: string, body: string, options?: RequestOptionsArgs):
  ➥ Observable<Response>;
  head(url: string, options?: RequestOptionsArgs): Observable<Response>;
}
```

Этот код написан на TypeScript, и у методов каждого `Http`-объекта есть обязательный аргумент `url`, который может быть либо `string`-, либо `Request`-объектом. Можно также передавать необязательный объект типа `RequestOptionArgs`. Каждый метод возвращает объект типа `Observable`, в который заключен объект типа `Response`.

Один из способов использования метода `get()` API `Http`-объекта, передающего URL в виде `string`-объекта, показан в следующем фрагменте кода:

```
constructor(private http: Http) {
  this.http.get('/products').subscribe(...);
}
```

Здесь не указан полный URL (такой как http://localhost:8000/products), поскольку предполагается, что Angular-приложение делает запрос к серверу по месту его развертывания, поэтому базовая часть веб-адреса может быть опущена. Метод `subscribe()` должен получить объект-наблюдатель с кодом для обработки полученных данных и ошибок.

Объект `Request` предлагает более универсальный API, позволяющий отдельно создавать `Request`-экземпляр, указывая HTTP-метод, и включать параметры поиска и `Header`-объект:

```
let myHeaders:Headers = new Headers();
myHeaders.append('Authorization', 'Basic QWxhZGRpb');
this.http
    .request(new Request({
      headers: myHeaders,
      method: RequestMethod.Get,
      url: '/products',
      search: 'zipcode=10001'
    }))
    .subscribe(...);
```

Объект `RequestOptionsArgs` объявлен как TypeScript-интерфейс:

```
export interface RequestOptionsArgs {
    url?: string;
    method?: string | RequestMethod;
    search?: string | URLSearchParams;
    headers?: Headers;
    body?: any;
    withCredentials?: boolean;
    responseType?: ResponseContentType;
}
```

Все элементы интерфейса являются необязательными, но если вы решите их применить, то компилятор TypeScript обеспечит предоставление вами значений правильного типа данных:

```
var myRequest: RequestOptionsArgs = {
    url: '/products',
```

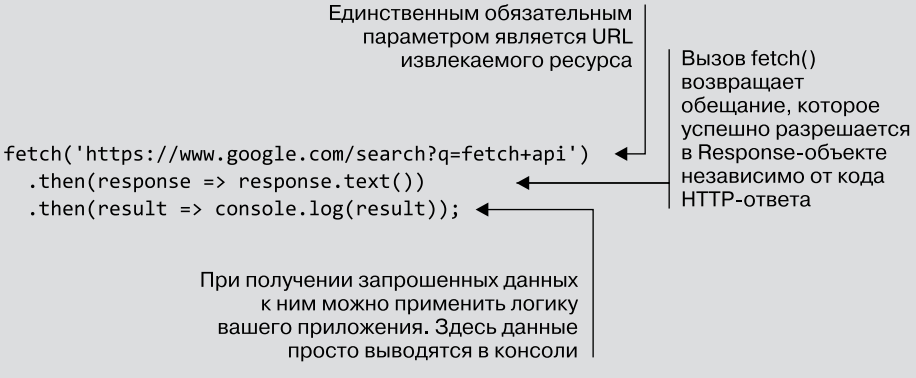
```
    method: 'Get'
};
this.http
    .request(new Request(myRequest))
    .subscribe(...)
```

В разделе «Практикум» будет показан пример использования свойства `search` объекта `RequestOptionsArgs` для создания HTTP-запроса, имеющего параметры строки запроса.

Что такое API Fetch

В настоящее время прилагаются усилия по унификации процесса сбора ресурсов в Интернете. Заменой объекту `XMLHttpRequest` может послужить API Fetch (https://fetch.spec.whatwg.org/). В нем определяются универсальные объекты `Request` и `Response`, которые могут применяться не только с HTTP, но также и с другими новыми веб-технологиями, например с Service Workers и Cache API.

При использовании API Fetch HTTP-запросы делаются с помощью глобальной функции `fetch()`:



Чтобы извлечь из ответа содержимое тела, нужно воспользоваться одним из методов `Response`-объекта. Каждым методом ожидается, что тело будет иметь вполне определенный формат. Тело считывается методом `text()` как обычный текст, который, в свою очередь, возвращает `Promise`-объект.

В отличие от имеющегося в Angular `Http`-сервиса на основе наблюдателей, API Fetch основан на промисах. Он упомянут в документации по Angular, поскольку им инспирировано несколько имеющихся в Angular классов и интерфейсов (например `Request`, `Response` и `RequestOptionsArgs`).

Чуть позже в этой главе будет показано, как выполнять запросы с помощью API `Http`-объекта и как обрабатывать HTTP-ответы путем подписки на наблюдаемые потоки. В главе 5 использовался сервер сервиса погоды, а здесь будет создан ваш собственный веб-сервер, задействующий среду Node.js.

8.2. Создание веб-сервера с применением Node и TypeScript

Вести разработку и развертывание веб-серверов позволяют многие платформы. В этой книге мы решили воспользоваться Node.js, руководствуясь следующими соображениями:

- чтобы разбираться в коде, не нужно учить новый язык программирования;
- Node позволяет создавать автономные приложения (такие как серверы);
- Node отлично справляется с задачами в области обмена данными, задействуя HTTP или WebSockets;
- применение Node позволяет продолжить написание кода в TypeScript, поэтому нам не придется объяснять, как создается веб-сервер на Java, .NET или Python.

Простой сервер в Node может быть написан всего лишь с помощью нескольких строчек кода, и вы начнете работы с самого простого варианта. Затем напишете веб-сервер, способный обслуживать данные в формате JSON (который, конечно же, будет применен в описаниях товаров) с использованием протокола HTTP. Чуть позже создадите еще одну версию сервера, способную вести обмен данными с клиентом через WebSocket-подключение. И наконец, в проекте, разрабатываемом в рамках практикума, мы научим вас создавать клиентскую часть аукциона, совершающую обмен данными с вашим веб-сервером.

8.2.1. Создание простого веб-сервера

В этом подразделе будет создано автономное Node-приложение, запускаемое в качестве сервера, поддерживающего примеры кода, созданные с помощью среды Angular. При обоюдной готовности серверной и клиентской сторон каталог проекта приобретет структуру, показанную на рис. 8.1.

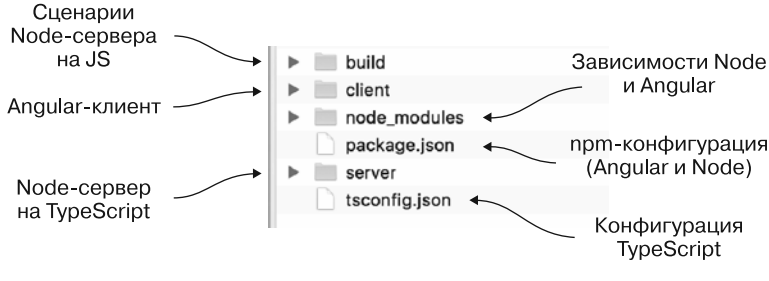


Рис. 8.1. Структура проекта Angular-Node-приложения

ПРИМЕЧАНИЕ

Если вы уже запустили примеры кода из приложения Б, то компилятор TypeScript на ваш компьютер установлен. Если же нет, то сделайте это сейчас.

Начнем с создания каталога по имени `http_websocket_samples` с подкаталогом `server`. Создаваемый здесь новый Node-проект следует настроить запуском следующей команды:

```
npm init -y
```

В главе 2 уже упоминалось, что ключ `-y` заставляет npm создать конфигурационный файл `package.json` с исходными установками, не выдавая никаких приглашений на выбор каких-либо вариантов.

Затем нужно создать файл `hello_server.ts`, имеющий следующее содержимое (листинг 8.1).

Листинг 8.1. hello_server.ts

Загружает Node-модуль с использованием ES6-синтаксиса import * as, который также поддерживается TypeScript

Этот простой сервер будет знать лишь, как отзываться с HTTP-статусом 200 и с текстом Hello World!, независимо от того, что именно запрашивается клиентом

```
import * as http from 'http';

    const server = http.createServer((request, response)=> {
    response.writeHead(200, {'Content-Type': 'text/plain'});
response.end('Hello World!\n');
});

const port = 8000;

server.listen(port);
console.log('Listening on http://localhost:' + port);
```

Функция listen() заставит данный сценарий выполняться бесконечно

Код в листинге 8.1 нуждается в транспиляции, поэтому для настройки tsc-компилятора в каталоге `http_websocket _samples` нужно создать файл `tsconfig.json` (листинг 8.2).

Листинг 8.2. Содержимое файла tsconfig.json

Заставляет tsc транспилировать модули в соответствии со спецификацией CommonJS. Инструкция (оператор) import в hello_server.ts будет транспилирована в код var http = require('http');

Транспилятор пометит файлы с расширением .js в каталог build

Компилятор tsc не станет транспилировать код, находящийся в каталоге node_modules

Чуть позже вы создадите каталог client, предназначенный для Angular-части приложения, но данный код не нуждается в транспиляции, поскольку SystemJS будет делать это динамически

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "outDir": "build"
  },
  "exclude": [
    "node_modules",
    "client"
  ]
}
```

После запуска команды `npm run tsc` в каталоге `build` будет сохранен транспилированный файл `hello_server.js`, и ваш веб-сервер можно будет запустить:

```
node build/hello_server.js
```

Node будет запущен с движком V8 JavaScript, который запустит сценарий из `hello_server.js`; он создаст веб-сервер и выведет в консоли следующее сообщение: Listening on http://localhost:8000. Если вы введете в свой браузер этот URL, то увидите веб-страницу с текстом Hello World!.

TypeScript 2.0 и @types

В этом проекте применяется установленный локально компилятор tsc версии 2.0, использующий пакеты @types для установки файлов, определяющих типы. Дело в том, что прежние версии tsc не поддерживают ключ компилятора types, и если у вас имеется глобально установленная устаревшая версия tsc, то она будет задействована при запуске tsc; это вызовет ошибки компиляции.

Чтобы убедиться в том, что используется локальная версия tsc, настройте ее в разделе `scripts` файла `package.json` командой (`"tsc": "tsc"`) и запустите компилятор, введя команду `npm run tsc` для транспиляции серверных файлов. Запустите данную команду из того же самого каталога, где находится файл `tsconfig.json` (из корневого каталога в примерах кода для этой главы).

Во избежание ошибок компиляции TypeScript для Node требуется наличие файлов определения типов (см. приложение Б). Чтобы установить необходимые Node определения типов для подобного проекта, нужно из корневого каталога вашего проекта запустить следующую команду:

```
npm i @types/node --save
```

Если используются примеры кода, предоставленные с этой главой, то можно запустить команду `npm install`, поскольку в файл `package.json` для Node включена зависимость `@types/node`:

```
"@types/node": "^4.0.30"
```

8.2.2. Обслуживание формата JSON

Во всех рассмотренных до сих пор примерах кода аукциона данные о товарах и обзорах были жестко заданы в файле `product-service.ts` в виде массивов объектов в формате JSON. В разделе «Практикум» эти данные будут перемещены на сервер, в связи с чем веб-серверу Node нужно знать порядок обслуживания формата JSON.

Чтобы отправить данные в формате JSON в браузер, нужно внести изменения в заголовок для указания MIME-типа `application/json`:

```
const server = http.createServer((request, response) => {
    response.writeHead(200, {'Content-Type': 'application/json'});
    response.end('{"message": "Hello Json!"}\n');
});
```

Этого фрагмента достаточно для иллюстрации отправки JSON-данных, однако на настоящих серверах выполняется большее количество функций, например чтение файлов, маршрутизация и обработка различных HTTP-запросов (`GET`, `POST` и т. д.). Чуть позже, в примере с аукционом, в зависимости от запроса вам придется откликаться, используя любые данные о товарах или обзоре.

Чтобы свести ручное программирование к минимуму, установим Express (http://expressjs.com), Node-среду, предоставляющую набор свойств, требуемых всеми веб-приложениями. Весь арсенал ее функциональных средств применять не придется, но она поможет создать веб-сервис на основе передачи состояния представления (RESTful), который будет возвращать данные в JSON-формате.

Для установки среды Express запустите из каталога `http_websocket_samples` следующую команду:

```
npm install express --save
```

При ее выполнении система Express будет загружена в папку `node_modules`, а в разделе `dependencies` файла `package.json` будут обновлены зависимости.

Поскольку в файле этого проекта есть запись `"@types/express": "^4.0.31"`, все типы определений для Express в вашем каталоге `node_modules` уже имеются. Но если нужно установить их в какой-либо другой проект, то запустите следующую команду:

```
npm i @types/express --save
```

Теперь систему Express можно импортировать в ваше приложение и приступить к использованию ее API при написании кода на TypeScript. В листинге 8.3 показано содержимое файла `my-express-server.ts`, в котором реализуется подпрограмма HTTP-запросов `GET` на стороне сервера.

Листинг 8.3. Содержимое файла my-express-server.ts

Создание экземпляра Express-объекта с использованием в качестве ссылки константы app

Маршрутизация реализуется только для GET-запросов, применяющих метод Express API get(), но в Express имеются методы, требуемые для обработки всего арсенала HTTP. Их объявления можно найти в файле express.d.ts

```
import * as express from "express";
const app = express();

app.get('/', (req, res) => res.send('Hello from Express'));

app.get('/products', (req, res) => res.send('Got a request for products'));

app.get('/reviews', (req, res) => res.send('Got a request for reviews'));

const server = app.listen(8000, "localhost", () => {
  const {address, port} = server.address();

  console.log('Listening on http://localhost:' + port);
});
```

Начало отслеживания на порте 8000 по адресу localhost и выполнение при запуске кода из функции, на которую указывает жирная стрелка

Для автоматического извлечения значений свойств address и port здесь используется деструктурирование (см. приложение А)

Если вместо деструктурирования задействован синтаксис ES5, то вместо одной строки кода придется применить две:

```
var address = server.address().address;
var port = server.address().port;
```

Транспилируйте содержимое файла `my-express-server.ts`, введя команду `npm run tsc`, и запустите этот сервер (`node build/my-express-server.js`). Как показано на рис. 8.2, в зависимости от веб-адреса, введенного в браузер, вы сможете запрашивать либо товары, либо сервисы.

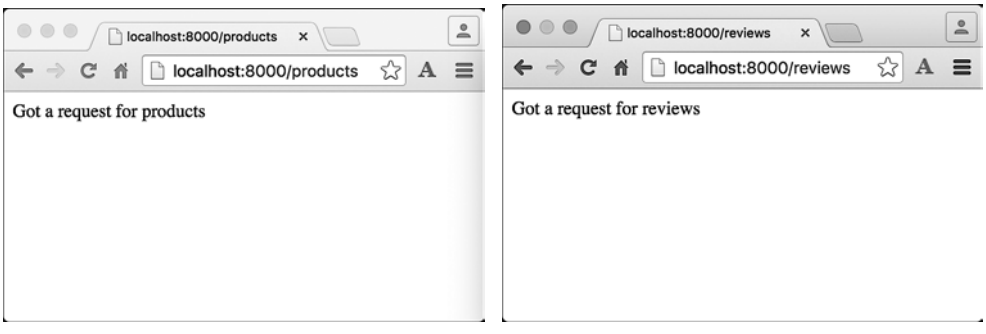


Рис. 8.2. Маршрутизация на стороне сервера с использованием Express

ПРИМЕЧАНИЕ

Для отладки Node-приложений обратитесь к предпочитаемой вами IDE-документации. Можно также прибегнуть к средству командной строки node-inspector (https://github.com/node-inspector/node-inspector).

8.2.3. Живая перекомпиляция TypeScript и перезагрузка кода

Примеры кода, работающего на стороне сервера, написаны на TypeScript, следовательно, прежде чем приступать к развертыванию кода в Node, для транспиляции этого кода в JavaScript нужно воспользоваться компилятором tsc. В подразделе Б.3.1 приложения Б будет рассмотрен ключ компиляции `-w`, с помощью которого tsc запускается в режиме отслеживания. Как только в файле TypeScript произойдут изменения, выполнится автоматическая перекомпиляция. Для включения режима автоматической компиляции вашего кода нужно, находясь в каталоге с исходными файлами, открыть отдельное командное окно и запустить в нем следующую команду:

```
tsc -w
```

Когда не указаны имена файлов, tsc для ключей компиляции задействует файл `tsconfig.json`. Теперь, как только вы внесете изменения в код TypeScript и сохраните файл, компилятор сгенерирует в каталоге `build`, как указано в файле

tsconfig.json, соответствующий файл с расширением .js. Следовательно, для запуска вашего веб-сервера с Node можно воспользоваться этой командой:

```
node build/my-express-server.js
```

Живая перекомпиляция кода TypeScript приносит реальную пользу, но Node-сервер не станет автоматически фиксировать изменения кода после своего запуска. Чтобы не пришлось перезапускать Node-сервер вручную с целью увидеть внесенные в код изменения в действии, можно воспользоваться весьма полезной утилитой Nodemon (http://nodemon.io). Она отследит любые изменения в вашем исходном коде и, как только они будут замечены, автоматически перезапустит сервер и перезагрузит код.

Утилиту Nodemon можно запустить либо глобально, либо локально. Для глобальной установки следует применять такую команду:

```
npm install -g nodemon
```

А эта команда запустит ваш сервер в режиме отслеживания:

```
nodemon build/my-express-server.js
```

Установите Nodemon локально (npm install nodemon --save-dev) и введите сценарии npm (https://docs.npmjs.com/misc/scripts) в файл package.json (листинг 8.4).

Листинг 8.4. Содержимое файла package.json

```
"scripts": {
  "tsc": "tsc",
  "start": "node build/my-express-server.js",
  "dev": "nodemon build/my-express-server.js"
},
"devDependencies": {
  "nodemon": "^1.8.1"
}
```

Пользуясь этими настройками, вы можете запустить сервер в режиме развертывания с помощью команды npm run dev (с автоматическим перезапуском и перезагрузкой) или в режиме prod-конфигурации через команду npm start (без перезапуска и перезагрузки). Мы присвоили команде запуска утилиты Nodemon имя dev, но вы можете выбрать имя по своему усмотрению, например startNodemon.

8.2.4. Добавление RESTful API для обслуживания товаров

Вашей конечной целью является обслуживание товаров и обзоров для приложения-аукциона. В этом подразделе будет показан способ подготовки Node-сервера, имеющего конечные точки REST, к получению HTTP-запросов GET, направляемых с целью обслуживания товаров, данные о которых имеют JSON-формат.

Код в файле my-express-server.ts будет изменен с целью обслуживания либо всех товаров, либо конкретно указанного одного товара (по его ID). Показанная далее измененная версия этого приложения находится в файле auction-rest-server.ts (листинг 8.5).

Листинг 8.5. Содержимое файла auction-rest-server.ts

```
import * as express from "express";
const app = express();

class Product {
    constructor(
        public id: number,
        public title: string,
        public price: number){}
    }

const products = [
    new Product(0, "First Product", 24.99),
    new Product(1, "Second Product", 64.99),
    new Product(2, "Third Product", 74.99)
];

function getProducts(): Product[] {
    return products;
}

app.get('/', (req, res) => {
    res.send('The URL for products is http://localhost:8000/products');
});

app.get('/products', (req, res) => {
    res.json(getProducts());
});

function getProductById(productId: number): Product {
    return products.find(p => p.id === productId);
}

app.get('/products/:id', (req, res) => {
    res.json(getProductById(parseInt(req.params.id)));
});

const server = app.listen(8000, "localhost", () => {
    const {address, port} = server.address();
    console.log('Listening on %s %s', address, port);
});
```

Определение класса Product

Создание массива из трех экземпляров Product с жестко заданными данными

Эта функция возвращает весь массив экземпляров Product

Возвращает текстовое приглашение в качестве ответа на GET-запрос, поступивший с базового URL

Когда Express получает GET-запрос, содержащий /products, он вызывает функцию getProducts() и возвращает клиенту результат в JSON-формате

Возвращает товар по его ID. Здесь показан новый метод Array.prototype.find(), появившийся в ES6. Если ваша IDE ничего не знает об этом методе, установите файл определения типа для полифилла ES6-shim: npminstall @types/es6-shim --save-dev

Когда Express получает GET-запрос с параметрами, их значения сохраняются в свойстве params объекта запроса. Вы конвертируете D товара из строки в целое число и вызываете метод getProductById(). Результат отправляется клиенту в JSON-формате

Теперь вы можете запустить в Node приложение `auction-rest-server.ts` (выполните команду `nodemon build/auction-rest-server.js`) и посмотреть, получает браузер все товары или выбранный товар. На рис. 8.3 показано окно браузера после ввода URL http://localhost:8000/products. Наш сервер возвращает все товары в JSON-формате.

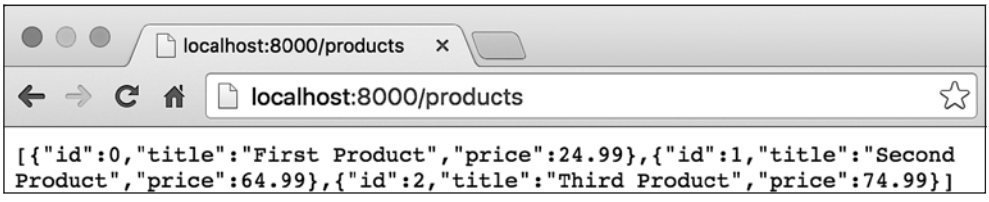


Рис. 8.3. Node-сервер отвечает на ввод адреса http://localhost:8000/products

На рис. 8.4. показано окно браузера после того, как в его адресную строку был введен URL http://localhost:8000/products/1. На этот раз сервером возвращены только данные о товаре, чей идентификатор имеет значение `1`.

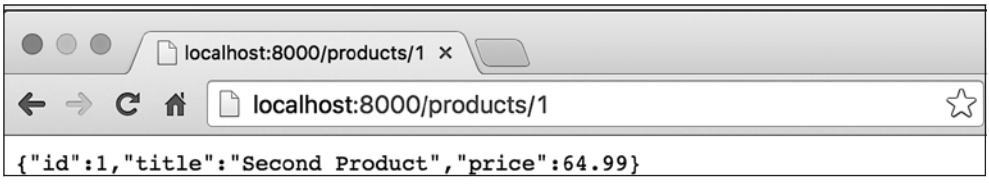


Рис. 8.4. Node-сервер отвечает на ввод адреса http://localhost:8000/products/1

Сервер готов. Теперь можно изучить способы инициирования HTTP-запросов и обработки ответов в Angular-приложениях.

8.3. Совместное использование Angular и Node

Ранее в настоящей главе была создана папка `http_websocket_samples`, содержащая файл `auction-rest-server.ts`. Это код Node-приложения, отвечающего на HTTP-запросы `GET` и предоставляющего сведения о товарах. В этом разделе будет создан Angular-клиент, который станет выдавать HTTP-запросы и обрабатывать сведения о товаре в качестве `Observable`-объекта, возвращенного вашим сервером. Код Angular-приложения будет находиться в подкаталоге `client` (см. рис. 8.1).

8.3.1. Статические ресурсы, имеющиеся на сервере

Обычное веб-приложение, развернутое на сервере, включает статические ресурсы (такие как код HTML, изображения, код CSS и код JavaScript), которые должны быть загружены в браузер, когда пользователь вводит веб-адрес приложения. Поскольку нами используется среда SystemJS, выполняющая транспиляцию динамически, допускается также присутствие в качестве статических ресурсов и файлов TypeScript.

С точки зрения Node та часть этого приложения, которая относится к Angular, рассматривается как статические ресурсы. Поскольку Angular-приложения загружают зависимости из `node_modules`, данный каталог тоже принадлежит к статическим ресурсам, востребуемым браузером.

В среде Express имеется специальный API для указания каталогов со статическими ресурсами, и вы внесете небольшие изменения в файл `auction-rest-server.ts`, показанный в листинге 8.5. В этом файле вы не станете указывать такой каталог, поскольку здесь не было развернуто какое-либо клиентское приложение. Новая версия данного файла будет называться `auction-rest-server-angular.ts`. Сначала добавьте следующие строки:

```
import * as path from "path";
app.use('/', express.static(path.join(__dirname, '..', 'client')));
app.use('/node_modules', express.static(path.join(__dirname, '..',
➥ 'node_modules')));
```

Когда браузер запрашивает статические ресурсы, Node ищет их в каталогах `client` и `node_modules`. Здесь, чтобы гарантировать создание путевого имени файла на кроссплатформенной основе, вы воспользуетесь имеющимся в Node API `path.join`. Его можно применять, когда нужно выстроить абсолютный путь к указанному файлу; примеры будут показаны чуть позже.

Сохраним на сервере те же самые конечные точки REST:

- `/` служит в качестве `main.html`, то есть начальной страницы приложения;
- `/products` предоставляет все товары;
- `/products/:id` предоставляет товар по его идентификатору ID.

В отличие от приложения `my_express_server.ts`, здесь вам не нужно, чтобы Node обрабатывал базовый URL. Он должен отправлять файл `main.html` в браузер. Измените в файле `auction-rest-server-angular.ts` маршрут для базового URL `/`, чтобы он приобрел следующий вид:

```
app.get('/', (req, res) => {
  res.sendFile(path.join(__dirname, '../client/main.html'));
});
```

Теперь, когда пользователь вводит в браузере веб-адрес Node-сервера, сначала обслуживается файл `main.html`. Затем сервер загружает ваше Angular-приложение со всеми зависимостями.

Общий конфигурационный файл NPM. В новой версии файла `package.json` (листинг 8.6) будут сочетаться все зависимости, требуемые как для кода, имеющего отношение к Node, так и для вашего Angular-приложения. Обратите внимание на то, что в разделе `scripts` объявляются несколько команд. Первая предназначена для запуска локально установленного компилятора tsc, а другие нужны для запуска Node-серверов для примеров кода, включенных в эту главу.

Листинг 8.6. Модифицированный файл package.json

```
{
  "private": true,
  "scripts": {
    "tsc": "tsc",
    "start": "node build/my-express-server.js",
    "dev": "nodemon build/my-express-server.js",
```

```
    "devRest": "nodemon build/auction-rest-server.js",
    "restServer": "nodemon build/auction-rest-server-angular.js",
    "simpleWsServer": "node build/simple-websocket-server.js",
    "twowayWsServer": "nodemon build/two-way-websocket-server.js",
    "bidServer": "nodemon build/bids/bid-server.js"
  },
  "dependencies": {
    "@angular/common": "^2.0.0",
    "@angular/compiler": "^2.0.0",
    "@angular/core": "^2.0.0",
    "@angular/forms": "^2.0.0",
    "@angular/http": "^2.0.0",
    "@angular/platform-browser": "^2.0.0",
    "@angular/platform-browser-dynamic": "^2.0.0",
    "@angular/router": "^3.0.0",
    "core-js": "^2.4.0",
    "rxjs": "5.0.0-beta.12",
    "systemjs": "0.19.37",
    "zone.js": "0.6.21",
    "@types/express": "^4.0.31",
    "@types/node": "^4.0.30",
    "express": "^4.14.0",
    "ws": "^1.1.1"
  },
  "devDependencies": {
    "@types/es6-shim": "0.0.30",
    "@types/ws": "0.0.29",
    "nodemon": "^1.8.1",
    "typescript": "^2.0.0"
  }
}
```

Обратите внимание: здесь включен пакет `@angular/http`, в котором содержится поддержка HTTP-протокола со стороны Angular. Кроме того, включены `ws` и `@types/ws` — они понадобятся в этой главе чуть позже, когда дело дойдет до поддержки WebSocket.

npm-сценарии

Менеджер пакетов npm поддерживает свойство `scripts` в `package.json` с более чем десятком сценариев, доступных в готовом виде (подробности можно найти в документации npm-scripts на https://docs.npmjs.com/misc/scripts). Применительно к вашей специфике разработки и развертывания вы можете добавить и новые команды.

Одни из этих сценариев нужно запускать вручную (например, `npm start`), а другие вызываются автоматически (например, `postinstall`). Как правило, если какая-либо команда в разделе `scripts` начинается с префикса `post`, то она будет запущена автоматически после той команды, которая указана после данного префикса. Например, в случае определения команды `i "postinstall" : "myCustomIstall.js"` при каждом запуске `npm install` будет запускаться и сценарий `myCustomIstall.js`.

Точно так же, если у команды имеется префикс `pre`, она будет запущена до запуска команды, названной после этого префикса. Например, в подразделе 10.3.2 в файле `package.json` вы увидите следующие команды:

```
"prebuild": "npm run clean && npm run test",
 "build": "webpack --config webpack.prod.config.js --progress --profile --
➥ colors"
```

Если запустить команду `build`, то npm сначала запустит сценарий, определенный в `prebuild`, а затем — сценарий, определенный в `build`.

До сих пор вы использовали только две команды: `npm start` и `npm run dev`. Но можете добавить к разделу `scripts` своего файла `package.json` какие угодно команды. Например, обе команды в предыдущем примере, и `build`, и `prebuild`, были из категории команд, определенных пользователем.

Сравнение общих и раздельных конфигурационных файлов

Все примеры кода в этой главе для клиента и сервера принадлежат единому npm-проекту и совместно задействуют один и тот же файл `package.json`. Все зависимости и типизация применяются клиентскими и серверными приложениями совместно. Подобная настройка может сэкономить время для установки зависимостей и пространство на диске, поскольку некоторые из зависимостей могут совместно использоваться как клиентом, так и сервером.

Но содержание кода для клиента и сервера в едином проекте может стать причиной усложнения процесса автоматической сборки в силу двух обстоятельств:

- клиенту и серверу могут понадобиться конфликтующие между собой версии конкретной зависимости;
- вы используете средство автоматической сборки, которому могут потребоваться различные конфигурации для клиента и сервера, и их каталоги `node_modules` не будут находиться в корневом каталоге проекта.

В главе 10 вам придется разделить клиентскую и серверную части онлайн-аукциона на два независимых npm-проекта.

Следующим шагом станет добавление Angular-приложения в каталог `client`.

8.3.2. Выполнение GET-запросов с помощью Http-объекта

Когда имеющийся в Angular `Http`-объект выполняет запрос, ответ возвращается в виде `Observable`-объекта, и клиентский код будет обрабатывать его, используя метод `subscribe()`. Начнем с простого приложения (`client/app/main.ts`), которое извлекает все товары из Node-сервера и выводит их, задействуя неупорядоченный HTML-список (листинг 8.7).

Листинг 8.7. Содержимое файла client/app/main.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule, Component } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import {HttpModule, Http} from '@angular/http';
import { Observable} from "rxjs/Observable";
import 'rxjs/add/operator/map';

@Component({
  selector: 'http-client',
  template: `<h1>All Products</h1>
  <ul>
    <li *ngFor="let product of products">
      {{product.title}}
    </li>
  </ul>
`})

class AppComponent {

  products: Array = [];

    theDataSource: Observable;

    constructor(private http: Http) {

        this.theDataSource = this.http.get('/products')
            .map(res => res.json());
    }

    ngOnInit(){

        // Получение данных с сервера
        this.theDataSource.subscribe(
            data => {
                if (Array.isArray(data)){
                    this.products=data;
                } else{
                    this.products.push(data);
                }
            },
            err =>
                console.log("Can't get products. Error code: %s, URL: %s ",
                ➥ err.status, err.url),
            () => console.log('Product(s) are retrieved')
        );

    }
}

@NgModule({
    imports:      [ BrowserModule,
```

Импортирование HTTP-модуля и Http-объекта, которые будут внедрены в AppComponent

В RxJS более 100 операторов, но в этом примере вам понадобится только map()

Экземпляр Http-сервиса, внедренного в компонент

Оператор map() конвертирует данные в JSON-строку и возвращает Observable-объект. Пока не вызван метод subscribe(), никакие запросы к серверу не выполняются

Метод subscribe() инициирует запрос к серверу. Внутри subscribe() создается Observer-объект, и это выражение, использующее жирную стрелку, присваивает полученные данные массиву products

Функция обратного вызова error вызывается только в том случае, если с сервера пришел ответ с ошибкой

Завершающая функция обратного вызова вызывается после того, как будет закончена обработка потока данных

```
                    HttpModule],
    declarations: [ AppComponent],
    bootstrap:    [ AppComponent ]
})
class AppModule { }

platformBrowserDynamic().bootstrapModule(AppModule);
```

Объявляет HttpModule, который определяет поставщиков, необходимых для внедрения Http-объекта

Чтобы увидеть функцию обратного вызова `error` в действии, измените конечную точку с `'/products'` на что-нибудь другое. Ваше Angular-приложение выведет в консоли следующее сообщение: Can't get products. Error code: 404, URL: http://localhost:8000/products.

ПРИМЕЧАНИЕ

HTTP-запрос GET отправляется на сервер, только когда вызывается метод subscribe(), и не отправляется при вызове метода get().

Теперь уже можно запустить сервер и ввести его URL в браузер, чтобы увидеть Angular-приложение в работе. Запустить Node-сервер можно, либо написав длинную команду:

```
node build/auction-rest-server-angular.js
```

либо используя npm-сценарий, который был определен вами в файле `package.json`:

```
npm run restServer
```

Откройте браузер, указав в нем адрес http://localhost:8000, и увидите Angular-приложение, показанное на рис. 8.5.

Рис. 8.5. Извлечение всех продуктов из Node-сервера

ПРИМЕЧАНИЕ

Убедитесь в том, что в файле client/systemjs.config.js пакет app отображается на main.ts.

СОВЕТ

Можно выполнить HTTP-запрос GET, передающий параметры в URL после знака вопроса (например, myserver.com?param1=val1&param2=val2). Метод Http.get() может принять второй параметр, являющийся объектом, реализующим RequestOptionsArgs. Поле search, имеющееся у RequestOptionsArgs, применяется для установки в качестве его значения либо строки, либо URLSearchParams-объекта. Пример использования URLSearchParams будет показан в разделе «Практикум».

8.3.3. Распаковка наблюдаемых объектов в шаблонах с помощью AsyncPipe

В предыдущем подразделе велась работа с наблюдаемым потоком товаров в коде TypeScript путем вызова метода `subscribe()`. Angular предлагает альтернативный синтаксис, рассмотренный в главе 5 и позволяющий обрабатывать наблюдаемые объекты прямо в шаблоне компонента с помощью каналов (pipes).

Angular включает канал `AsyncPipe` (или `async` при использовании в шаблонах), который может получать в качестве ввода `Promise`- или `Observable`-объект и подписываться на него в автоматическом режиме. Чтобы увидеть все это в действии, внесем в код из предыдущего подраздела следующие изменения:

- изменим тип переменной `products` с `Array` на `Observable`;
- удалим объявление переменной `theDataSource`;
- удалим из кода вызов `subscribe()`. Объект `Observable`, который возвращается `http.get().map()`, будет присваиваться переменной `products`;
- добавим в шаблоне канал `async` к циклу `*ngFor`.

Все эти изменения реализованы в следующем коде (`main-asyncpipe.ts`) (листинг 8.8).

Листинг 8.8. Содержимое файла main-asyncpipe.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule, Component } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import {HttpModule, Http} from '@angular/http';
import 'rxjs/add/operator/map';
import { Observable} from "rxjs/Observable";

@Component({
  selector: 'http-client',
  template: `<h1>All Products</h1>
  <ul>
    <li *ngFor="let product of products | async">
       {{product.title}}
    </li>

  </ul>
  `})
class AppComponent {

  products: Observable<Array<string>>;

  constructor(private http: Http) {

    this.products = this.http.get('/products')
        .map(res => res.json());
  }
}
```

Канал async распаковывает элементы массива из предоставленного наблюдаемого потока товаров

Теперь массив товаров products имеет тип Observable, в который заключен массив строк

Присваивание свойству products объекта Observable, который будет возвращен методом map()

```
@NgModule({
  imports: [ BrowserModule, HttpModule],
  declarations: [ AppComponent],
  bootstrap: [ AppComponent ]
})
class AppModule { }

platformBrowserDynamic().bootstrapModule(AppModule);
```

Вывод при запуске этого приложения будет таким же, как и на рис. 8.5.

ПРИМЕЧАНИЕ

Эта версия AppComponent с async короче версии, показанной в листинге 8.7. Но код, в котором subscribe() вызывается явным образом, проще тестировать.

8.3.4. Внедрение HTTP в сервис

В этом подразделе мы покажем пример внедряемого класса `ProductService`, в котором будет инкапсулироваться обмен данными с сервером по протоколу HTTP. Вы создадите небольшое приложение, в котором пользователь сможет вводить идентификатор товара и заставлять приложение выполнять запрос к конечной точке сервера `/products/:id`.

Пользователь вводит идентификатор товара и нажимает кнопку, запуская тем самым подписку на свойство по имени `productDetails` объекта типа `Observable`, принадлежащее объекту `ProductService`. На рис. 8.6 показаны внедряемые объекты приложения, которые вам предстоит создать.

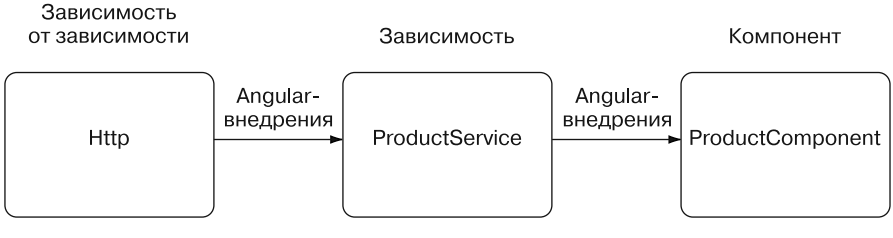


Рис. 8.6. Рабочий поток «клиент — сервер»

В главе 7 вы познакомились с API Forms. Здесь с помощью простой формы вы создадите компонент `AppComponent`, у которого есть поле ввода данных и кнопка поиска товара Find Product (Найти продукт). Это приложение станет обмениваться данными с ранее созданным веб-сервером Node, клиентская часть будет реализована в двух последовательных улучшениях. В первой версии (`main-form.ts`) (листинг 8.9) класс `ProductService` использоваться не будет. Компонент `AppComponent` получит внедренный `Http`-объект и выполнит запрос к серверу.

Листинг 8.9. Содержимое файла main-form.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule, Component } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule} from '@angular/forms';
import {HttpModule, Http} from '@angular/http';

@Component({
  selector: 'http-client',
  template: `<h1>Find Product By ID</h1>
    <form #f="ngForm" (ngSubmit) = "getProductByID(f.value)" >
      <label for="productID">Enter Product ID</label>
      <input id="productID" type="number" name = "productID" ngModel>
      <button type="submit">Find Product</button>
    </form>

    <h4>{{productTitle}} {{productPrice}}</h4>
  `})

class AppComponent {

  productTitle: string;
  productPrice: string;

  constructor(private http: Http) {}

  getProductByID(formValue){
    this.http.get(`/products/${formValue.productID}`)
        .map(res => res.json())
        .subscribe(
          data => {this.productTitle= data.title;
                   this.productPrice=`$` + data.price;},
          err => console.log("Can't get product details. Error code: %s,
          ➥ URL: %s ",
                err.status, err.url),
          () =>    console.log( 'Done')
        );
  }
}

@NgModule({
    imports: [ BrowserModule, FormsModule, HttpModule],
    declarations: [ AppComponent],
    bootstrap: [ AppComponent ]
})
class AppModule { }

platformBrowserDynamic().bootstrapModule(AppModule);
```

Определяет форму которая вызывает getProductByID (), когда пользователь нажимает кнопку Отправить

Для прикрепления введенного идентификатора товара roductID к веб-адресу в HTTP-запросе get() используется строковая интерполяция. HTTP-запрос GET в этом месте еще не выдается

При возникновении ошибки на консоль выводятся код ошибки и URL

Метод subscribe() выдает GET-запрос и присваивает полученные значения переменным класса productTitle и productPrice, которые привязаны к HTML-шаблону

На рис. 8.7 показан снимок экрана, сделанный после ввода в поле идентификатора товара цифры 2 и нажатия кнопки Find Product (Найти продукт), инициирующей отправку запроса по URL http://localhost:8000/products/2. Сервер Node Express

соотносит `/products/2` с соответствующей конечной точкой REST и направляет этот запрос к методу, определенному как `app.get('/products/:id')`.

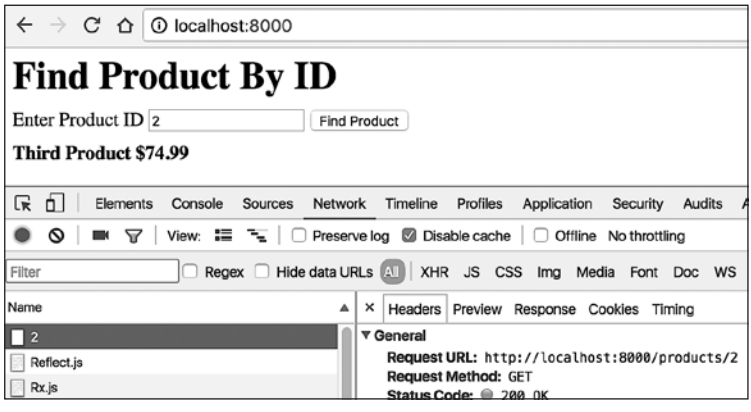


Рис. 8.7. Получение сведений о товаре по идентификатору

Теперь введем класс `ProductService` (`product-service.ts`). В листинге 8.9 `Http`-объект внедряется в конструктор `AppComponent`. Теперь нужно переместить код, использующий `Http`-объект в `ProductService`, чтобы код стал отражением показанной на рис. 8.6 архитектуры (листинг 8.10).

Листинг 8.10. product-service.ts

```
import { Http} from '@angular/http';
import { Injectable} from "@angular/core";
import {Observable} from 'rxjs/Observable';
import 'rxjs/add/operator/map';

@Injectable()
export class ProductService{

    constructor(private http: Http){}

    getProductByID(productID: string): Observable<any>{
       return this.http.get(`/products/${productID}`)
            .map(res => res.json());
    }
}
```

Angular внедряет экземпляр Http-объекта, а спецификатор private приводит к подразумеваемому созданию переменной экземпляра http

Метод getProductByID() составляет URL, но не вызывает метод subscribe(). Он возвращает Observable-объект. Компонент, обрабатывающий данные, будет предоставлен наблюдателем

В предыдущих версиях ProductService аннотация @Injectable не использовалась, потому что в сам ProductService ничего не внедрялось

Класс `ProductService` использует внедрение зависимости (DI). Декоратор `@Injectable()` заставляет компилятор TypeScript создать для `ProductService` метаданные, и применять данный декоратор здесь необходимо. Когда вы внедряли `Http`-объект в компонент, имеющий еще один декоратор (`@Component`), это

был сигнал компилятору TypeScript создать метаданные для компонента, требуемого для DI. Если в классе `ProductService` не имелось никаких декораторов, то компилятор TypeScript не станет создавать для него никаких метаданных и механизм Angular DI не узнает, что ему следует выполнить какое-то внедрение в `ProductService`. Для классов, представляющих сервисы, требуется простое присутствие декоратора `@Injectable()`, и вы не должны забывать включать в файле `tsconfig.json` настройку `"emitDecoratorMetadata": true`.

Подписчиком на наблюдаемый поток, производимый `ProductService`, станет новая версия `AppComponent` (`main-with-service.ts`) (листинг 8.11).

Листинг 8.11. Содержимое файла main-with-service.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule, Component } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule} from '@angular/forms';
import {HttpModule, Http} from '@angular/http';
import {ProductService} from './product-service';

@Component({
  selector: 'http-client',
  providers: [ProductService],
  template: `<h1>Find Product By ID Using ProductService</h1>
     <form #f="ngForm" (ngSubmit)="getProductByID(f.value)">
       <label for="productID">Enter Product ID</label>
       <input id="productID" type="number" ngControl="productID">
       <button type="submit">Find Product</button>
     </form>
     <h4>{{productTitle}} {{productPrice}}</h4>
  `})
class AppComponent {

  productTitle: string;
  productPrice: string;

  constructor(private productService: ProductService) {}

  getProductByID(formValue){
      this.productService.getProductByID(formValue.productID)
        .subscribe(
            data => {this.productTitle = data.title;
                     this.productPrice = `$` + data.price;},
            err => console.log("Can't get product details.
            ➥ Error code: %s, URL: %s ",
                  err.status, err.url),
      () => console.log('Done')
  );
}

}
@NgModule({
      imports: [ BrowserModule, FormsModule, HttpModule],
      declarations: [ AppComponent],
```

Теперь Angular внедряет ProductService, тогда как в предыдущей версии этого компонента внедрялся Http-объект

Вызывает метод ProductService, возвращающий Observable-объект

Подписывается на Observable и обрабатывает результаты

```
      bootstrap: [ AppComponent ]
})
class AppModule { }

platformBrowserDynamic().bootstrapModule(AppModule);
```

Здесь `ProductService` является не компонентом, а классом, и Angular не позволяет указывать для классов поставщиков. В результате поставщик указывается для `Http`-объекта в `AppComponent` путем включения свойства `providers` в декоратор `@Component`. Другой возможный вариант может заключаться в объявлении поставщиков в `@NgModule`. В данном конкретном приложении это ни на что не повлияет.

В главе 4 в процессе обсуждения DI мы упоминали о способности Angular внедрять объекты, и если у них имеются свои собственные зависимости, то Angular займется и их внедрением. В листинге 8.11 доказывается, что имеющийся в Angular модуль DI работает в точном соответствии с нашими ожиданиями.

8.4. Обмен данными между клиентом и сервером через веб-сокеты

WebSocket является двоичным протоколом с низким уровнем издержек, поддерживаемым всеми современными браузерами. Как показано на рис. 8.8, при использовании HTTP-протокола, основанного на запросах, клиент отправляет запрос на соединение с сервером и ожидает поступления ответа (полудуплексный режим связи). А как показано на рис. 8.9, протокол WebSocket позволяет данным одновременно путешествовать по соединению в обоих направлениях (полнодуплексный режим связи). WebSocket-соединение постоянно поддерживается в рабочем состоянии, что дает дополнительные преимущества: низкое значение задержки во взаимодействии сервера и клиента.

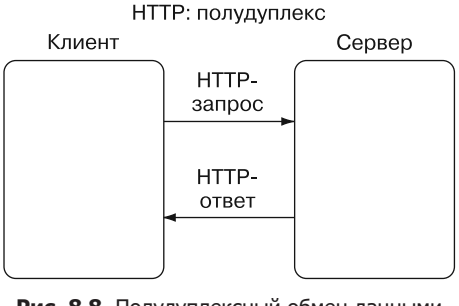


Рис. 8.8. Полудуплексный обмен данными

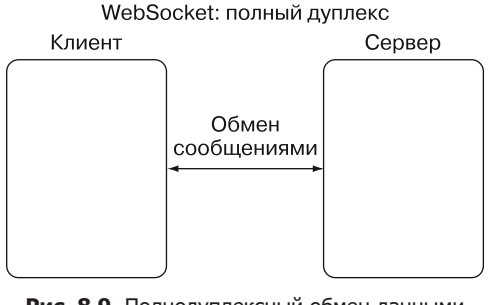


Рис. 8.9. Полнодуплексный обмен данными

По сравнению с обычным HTTP-протоколом типа «запрос — ответ», добавляющим к данным приложения несколько сотен байт (в заголовках), издержки при использовании веб-сокетов составляют всего лишь пару байт. Если вам не знакома эта технология, то обратитесь к ресурсу www.websocket.org или к одному из множества руководств, доступных в Интернете.

8.4.1. Выдача данных с Node-сервера

Веб-сокеты поддерживаются большинством платформ, работающих на стороне сервера (Java, .NET, Python и др.), но вы для реализации вашего сервера на основе веб-сокета продолжите работать с платформой Node. Вы реализуете один конкретный вариант применения: сервер будет выдавать данные клиенту, использующему браузер, как только клиент подключится к сокету. Мы намеренно не станем отправлять запрос данных от клиента, для демонстрации того, что веб-сокеты не работают по схеме «запрос — ответ». Приступить к отправке данных по веб-сокет-соединению может любая из сторон.

WebSocket-протокол реализуется несколькими Node-пакетами, но здесь будет использоваться npm-пакет `ws` (https://www.npmjs.com/package/ws). Установите этот пакет, подав следующую команду, находясь в каталоге вашего проекта:

```
npm install ws --save
```

Затем установите для `ws` файл определения типов:

```
npm install @types/ws --save-dev
```

Теперь компилятор TypeScript не станет возражать при использовании API из пакета `ws`. Кроме этого, данный файл пригодится для просмотра доступных API и типов.

Ваш первый WebSocket-сервер будет предельно прост: он станет выдавать клиенту текст This message was pushed by the WebSocket server, как только будет установлено соединение. Вы намеренно не хотите, чтобы клиент отправлял на сервер какой-либо запрос, для демонстрации того, что сокет является «улицей с двухсторонним движением» и сервер может выдавать данные без церемонии запроса.

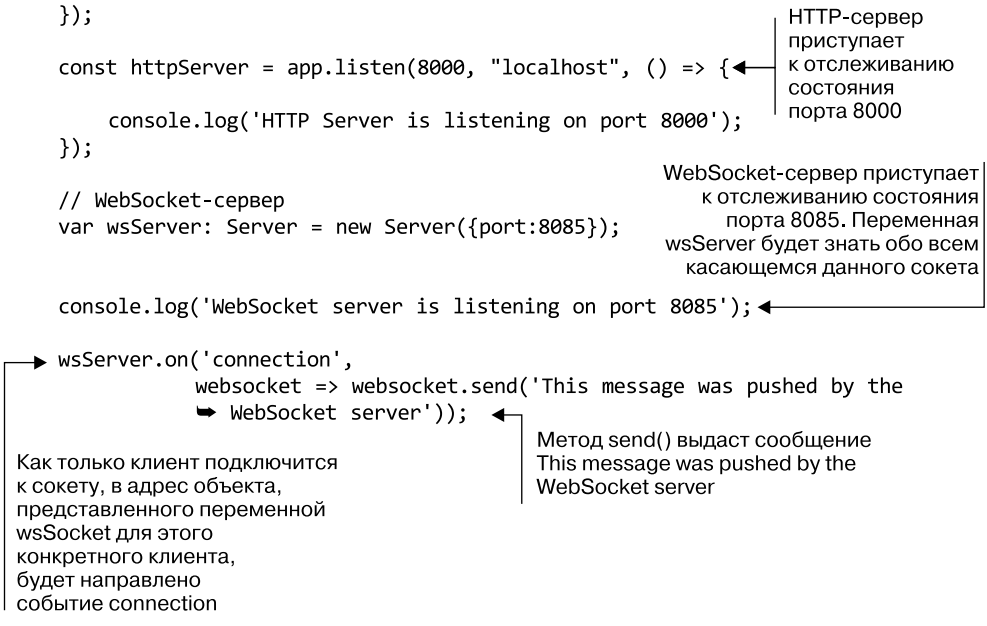
Приложение, показанное в листинге 8.12 (`simple-websocket-server.ts`), создает два сервера. HTTP-сервер будет запущен с использованием порта 8000 и станет отвечать за отправку клиенту исходного HTML-файла. А WebSocket-сервер будет запущен с применением порта 8085 и станет обмениваться данными со всеми подключившимися клиентами через этот порт.

Листинг 8.12. simple-websocket-server.ts

```
import * as express from "express";
import * as path from "path";
import {Server} from "ws";

const app = express();

app.use('/', express.static(path.join(__dirname, '..', 'client')));
app.use('/node_modules', express.static(path.join(__dirname, '..',
➥ 'node_modules')));

// HTTP-сервер
app.get('/', (req, res) => {
    res.sendFile(path.join(__dirname, '..',
    ➥ 'client/simple-websocket-client.html'));
```

В данном примере для явного объявления переменных используется тип Server из модуля ws. Именно поэтому импортируется только определение Server

Как только HTTP-клиент подключится к базовому URL, HTTP-сервер отправит обратно файл client/simple-websocket-client.html



ПРИМЕЧАНИЕ

В листинге 8.12 из ws импортируется только модуль Server. При использовании других экспортируемых элементов можно задействовать запись import * as ws from "ws"; .

В листинге 8.12 серверы HTTP и WebSocket запущены на разных портах, но можно повторно применять один и тот же порт, предоставив конструктору `WsServer` заново созданный экземпляр класса `httpServer`:

```
const httpServer = app.listen(8000, "localhost", () => {...});
const wsServer: WsServer = new WsServer({server: httpServer});.
```

В разделе «Практикум» порт 8000 будет использоваться для обоих протоколов обмена данными: и HTTP, и WebSocket (см. файл `server/auction.ts`).

ПРИМЕЧАНИЕ

Как только к серверу подключится новый клиент, ссылка на это соединение добавляется к массиву wsServer.clients, чтобы сообщения при необходимости можно было распространять среди всех подключенных клиентов: wsServer.clients.forEach (client => client.send(`...`)); .

Содержимое клиентского файла `simple-websocket-client.html` показано в листинге 8.13. Этот клиент не применяет ни Angular, ни TypeScript. Как толь-

ко данный файл загружается в браузер, его сценарий подключается к вашему WebSocket-серверу по адресу ws://localhost:8085. Учтите, что протоколом является `ws`, а не `http`. Для безопасного сокет-подключения следует воспользоваться протоколом `wss`.

Листинг 8.13. simple-websocket-client.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
</head>
<body>
<span id="messageGoesHere"></span>

<script type="text/javascript">
    var ws = new WebSocket("ws://localhost:8085");

    ws.onmessage = function(event) {
        var mySpan = document.getElementById("messageGoesHere");
        mySpan.innerHTML=event.data;
    };

    ws.onerror = function(event){
        console.log("Error ", event)
    }
</script>
</body>
</html>
```

Устанавливает подключение с сокетом. На этом этапе сервер обновляет протокол от HTTP до WebSocket

Когда сообщение поступает с сокета, вы выводите его содержимое в элемент

В случае ошибки журналы браузера выводят в консоли сообщение

Для запуска сервера, выдающего данные клиенту, запустите Node-сервер (`node build/simple-websocket-server.js` или `npm simpleWsServer`). Он выведет в консоли следующее сообщение:

```
WebSocket server is listening on port 8085
HTTP Server is listening on 8000
```

ПРИМЕЧАНИЕ

При изменении кода, находящегося в каталоге server, не забудьте запустить команду npm run tsc в корневом каталоге своего проекта, чтобы создать свежую версию вашего кода JavaScript в каталоге build. В противном случае команда node загрузит старый файл JavaScript.

Чтобы получить сообщение, выданное сервером, откройте в браузере страницу http://local-host:8000. Будет выведено сообщение, показанное на рис. 8.10.

В данном примере HTTP-протокол используется только для начальной загрузки HTML-файла. Затем клиент запрашивает обновление протокола до WebSocket (код состояния 101), и с этого момента приложение уже не будет применять протокол HTTP.

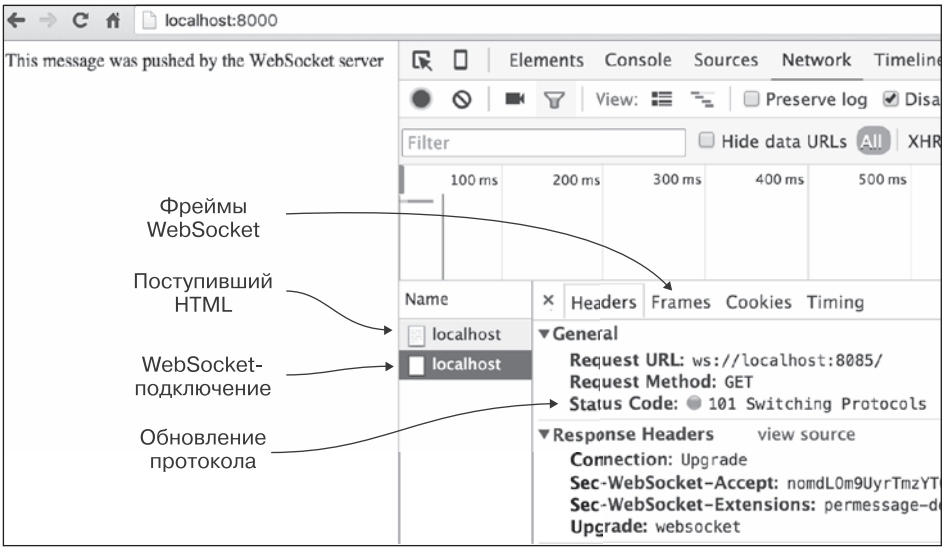


Рис. 8.10. Получение сообщения от сокета

СОВЕТ

Отслеживать сообщения, проходящие через сокет, можно с помощью вкладки Frames в Developer Tools (Инструменты разработчика) браузера Chrome.

8.4.2. Превращение WebSocket в наблюдаемый объект

В предыдущем подразделе был создан код клиента на JavaScript (без Angular) с использованием принадлежащего браузеру `WebSocket`-объекта. Теперь будет показано, как создается сервис, заключающий браузерный объект `WebSocket` в наблюдаемый поток, чтобы Angular-компоненты могли подписаться на сообщения, поступающие от сервера через сокет-соединение.

Ранее, в подразделе 8.3.2, код, получающий данные о товаре, был структурирован следующим образом (в псевдокоде):

```
this.http.get('/products')
  .subscribe(
      data => handleNextDataElement(),
      err => handleErrors(),
      () => handleStreamCompletion()
);
```

По сути, вашей целью было написание кода приложения, задействующего наблюдаемый поток, предоставляемый имеющимся в Angular `Http`-сервисом. Но в данном фреймворке нет сервиса, который производил бы наблюдаемый объект из WebSocket-соединения, так что требуемый сервис придется создавать. Тогда Angular-клиент получит возможность подписываться на сообщения, приходящие от WebSocket-соединения, точно так же, как это делалось при использовании `Http`-объекта.

Заключение любого сервиса в наблюдаемый поток

Теперь вы создадите небольшое Angular-приложение, которое не будет использовать WebSocket-сервер, но покажет, как заключить логику функционирования в Angular-сервис, выдающий данные с помощью наблюдаемого потока. Начнем с создания наблюдающего сервиса, который будет выдавать жестко заданные значения без фактического подключения к сокету. В листинге 8.14 создается сервис, выдающий каждую секунду текущее время.

Листинг 8.14. custom-observable-service.ts

```
import {Observable} from 'rxjs/Rx';
export class CustomObservableService{
  createObservableService(): Observable<Date>{
      return new Observable(
          observer => {
              setInterval(() =>
                  observer.next(new Date())
              , 1000);
          }
      );
  }
}
```

Здесь создается наблюдаемый объект, исходя из предположения, что подписчик предоставит `Observer`-объект, который знает, что делать с данными, предоставленными наблюдаемым объектом. Как только наблюдаемый объект вызывает в отношении наблюдателя метод `next()`, подписчик получает значение, предоставленное в качестве аргумента (в данном примере это `new Date()`). Поток данных никогда не выдает ошибку и никогда не завершается.

ПРИМЕЧАНИЕ

Можно также создать подписчика на наблюдаемый объект путем явного вызова метода Subscriber.create(). Соответствующий пример будет показан в разделе «Практикум».

Компонент `AppComponent` в листинге 8.15 получает внедренный `CustomObservableService`, вызывает метод `createObservableService()`, который возвращает `Observable`-объект, и подписывается на него, создавая наблюдателя, знающего, что делать с данными. Наблюдатель в данном примере присваивает полученное время переменной `currentTime`.

Листинг 8.15. Содержимое файла custom-observable-service-subscriber.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule, Component }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import 'rxjs/add/operator/map';
import {CustomObservableService} from "./custom-observable-service";
```

```
@Component({
  selector: 'app',
  providers: [ CustomObservableService ],
  template: `<h1>Simple subscriber to a service</h1>
       Current time: {{currentTime | date: 'jms'}}
  `})
class AppComponent {
  currentTime: Date;
  constructor(private sampleService: CustomObservableService) {
      this.sampleService.createObservableService()
        .subscribe( data => this.currentTime = data );
  }
}
@NgModule({
  imports:      [ BrowserModule],
  declarations: [ AppComponent],
  bootstrap:    [ AppComponent ]
})
class AppModule { }
platformBrowserDynamic().bootstrapModule(AppModule);
```

Для этого приложения в корневом каталоге проекта будет создан файл `index.html`. В приложении не используются никакие серверы, и его можно запустить путем ввода команды `live-server` в окне терминала. Текущее время в окне браузера будет обновляться каждую секунду. Здесь применяется канал `DatePipe` с форматом `'jms'`, обеспечивающий отображение только часов, минут и секунд (все форматы данных описаны в документации Angular DatePipe на https://angular.io/api/common/DatePipe).

Это очень простой пример, но он демонстрирует основной прием для упаковки любой логики приложения в наблюдаемый поток и подписки на него. В данном случае используется метод `setInterval()`, но его можно заменить любым кодом конкретного назначения, создающим одно значение и более и отправляющим их в потоке.

Не забудьте про обработку ошибок и при надобности про завершение потока. В следующем фрагменте кода показывается простой наблюдаемый объект, отправляющий наблюдателю один элемент, способный выдать ошибку и сообщить наблюдателю о том, что поток завершен:

```
return new Observable(
    observer => {
      try {
        observer.next('Hello from observable');
       //throw ("Got an error");
          } catch(err) {
             observer.error(err);
          } finally{
            observer.complete();
          }
    }
);
```

Если в строку со `throw` не включить комментарий, то вызывается `observer.error()`, в результате чего в отношении подписчика вызывается обработчик при наличии такового.

Теперь научим Angular-сервис обмениваться данными с WebSocket-сервером.

Angular обменивается информацией с WebSocket-сервером

Создадим небольшое Angular-приложение с WebSocket-сервисом (на стороне клиента) который взаимодействует с Node WebSocket-сервером. Серверный уровень может быть реализован с применением любой технологии, поддерживающей веб-сокеты. Архитектура такого приложения показана на рис. 8.11.

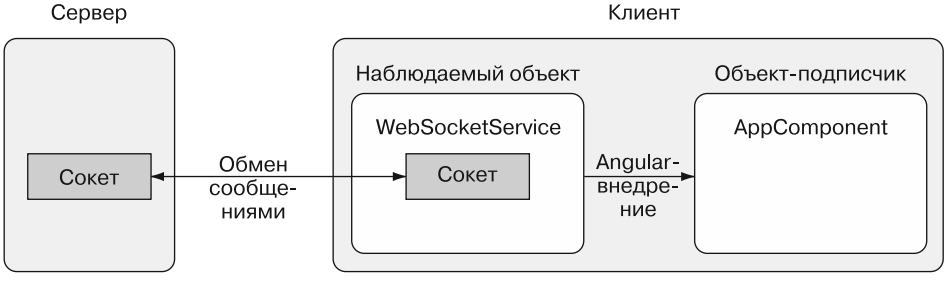


Рис. 8.11. Взаимодействие Angular-приложения с сервером с помощью сокета

Код в листинге 8.16 заключает `WebSocket`-объект браузера в наблюдаемый поток. Данный сервис создает экземпляр класса `WebSocket`, который подключается к серверу на основе предоставленного URL, и этот экземпляр обрабатывает сообщения, полученные с сервера. У объекта `WebSocketService` также имеется метод `sendMessage()`, позволяющий клиенту отправлять сообщения на сервер.

Листинг 8.16. Содержимое файла websocket-observable-service.ts

```
import {Observable} from 'rxjs/Rx';
export class WebSocketService{
    ws: WebSocket;
    createObservableSocket(url:string):Observable{
        this.ws = new WebSocket(url);
        return new Observable(
          observer => {
              this.ws.onmessage = (event) =>
                        observer.next(event.data);
              this.ws.onerror = (event) => observer.error(event);
              this.ws.onclose = (event) => observer.complete();
          }
      );
    }
    sendMessage(message: any){
         this.ws.send(message);
    }
}
```

ПРИМЕЧАНИЕ

В листинге 8.16 показан один из способов создания наблюдаемого объекта из WebSocket. В качестве альтернативы для получения того же результата можно воспользоваться методом Observable.webSocket().

В листинге 8.17 показан код компонента `AppComponent`, который подписывается на `WebSocketService`, внедряемый в `AppComponent`, представленный на рис. 8.11. Этот элемент может также отправлять сообщения серверу, когда пользователь нажимает кнопку Send Msg to Server (Отправить сообщение серверу).

Листинг 8.17. Содержимое файла websocket-observable-service-subscriber.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule, Component } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import {WebSocketService} from "./websocket-observable-service";
@Component({
  selector: 'app',
  providers: [ WebSocketService ],
  template: `<h1>Angular subscriber to WebSocket service</h1>
       {{messageFromServer}}<br>
       <button (click)="sendMessageToServer()">Send msg to Server</button>
  `})
class AppComponent {
  messageFromServer: string;
  constructor(private wsService: WebSocketService) {
      this.wsService.createObservableSocket("ws://localhost:8085")
        .subscribe(
            data => {
              this.messageFromServer = data;
            },
            err => console.log( err),
            () => console.log( 'The observable stream is complete')
        );
}
  sendMessageToServer(){
      console.log("Sending message to WebSocket server");
      this.wsService.sendMessage("Hello from client");
  }
}
@NgModule({
    imports:      [ BrowserModule],
    declarations: [ AppComponent],
    bootstrap:    [ AppComponent ]
})
class AppModule { }
platformBrowserDynamic().bootstrapModule(AppModule);
```

HTML-файл, выводящий этот компонент на экран, называется `two-way-websocket-client.html` (листинг 8.18). Нужно убедиться в том, что в качестве основного сценария приложения в `systemjs.config.js` фигурирует `websocket-observable-service-subscriber`.

Листинг 8.18. Содержимое файла two-way-websocket-client.html

```
<!DOCTYPE html>
<html>
<head>
  <title>Http samples</title>
  <script src="https://cdn.polyfill.io/v2/
    polyfill.js?features=Intl.~locale.en"></script>
  <script src="node_modules/zone.js/dist/zone.js"></script>
  <script src="node_modules/typescript/lib/typescript.js"></script>
  <script src="node_modules/reflect-metadata/Reflect.js"></script>
  <script src="node_modules/rxjs/bundles/Rx.js"></script>
  <script src="node_modules/systemjs/dist/system.src.js"></script>
  <script src="systemjs.config.js"></script>
  <script>
    System.import('app').catch(function (err) {console.error(err);});
  </script>
</head>
<body>
<app>Loading...</app>
</body>
</html>
```

И наконец, нужно создать еще одну версию `simple-websocket-server.ts`, чтобы обслуживать HTML-файл с другим Angular-клиентом. Этот сервер будет реализован в файле `two-way-websocket-server.ts`, и в нем будет практически такой же код с двумя небольшими изменениями (представлены ниже).

1. Когда сервер получает запрос к базовому URL, возникает необходимость обслуживания предыдущего HTML для клиента:

   ```
   app.get('/', (req, res) => { res.sendFile(path.join(__dirname, '..',
   ➥ 'client/two-way-websocket-client.html'));
   });
   ```

2. Для обработки сообщений, поступающих от клиента, нужно добавить обработчик `on('message')`:

   ```
   wsServer.on('connection',
     websocket => {
         websocket.send('This message was pushed by the WebSocket server');
         websocket.on('message',
                      message => console.log("Server received: %s",
                      ➥ message));
         });
   ```

Чтобы увидеть это приложение в работе, запустите `nodemon build/two-way-websocket_server.js` (или воспользуйтесь командой `npm run twowayWsServer`, сконфигурированной в файле `package.json`) и откройте браузер по адресу localhost:8000. Будет выведено окно с сообщением, выданным из Node, и если нажать кнопку, то на сервер будет отправлено сообщение Hello from client. Снимок экрана, показанный на рис. 8.12, был сделан после однократного нажатия кнопки (в инструментах разработчика (Chrome Developer Tools) была открыта вкладка Frames, находящаяся во вкладке Network (Сеть)).

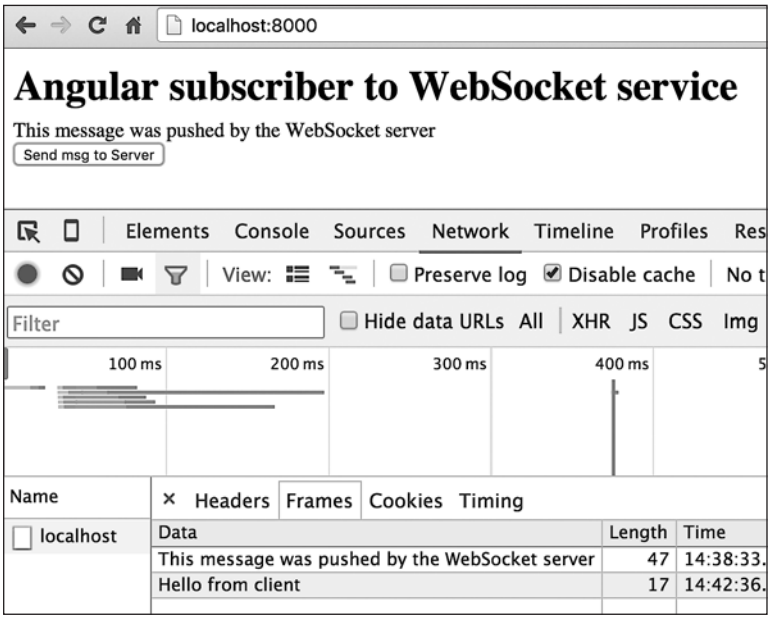


Рис. 8.12. Получение сообщения в Angular от Node

Теперь, усвоив порядок обмена данными с сервером через протоколы HTTP и WebSocket, научим онлайн-аукцион взаимодействовать с Node-сервером.

8.5. Практикум: реализация поиска товара и уведомлений о ценовых предложениях

В версию аукциона этой главы добавлен весьма значительный объем кода, так что мы решили избавить вас от его ввода. В данном практикуме будут рассмотрены только новые и измененные фрагменты, присутствующие в новой версии приложения-аукциона, созданные при изучении текущей главы. Эта версия приложения нацелена на решение двух основных задач.

- Реализация функциональных возможностей по поиску товара. Компонент `SearchComponent` подключит аукцион к Node-серверу через HTTP, а сведения о товарах и обзоры будут поступать с сервера.
- Добавление выдаваемых сервером уведомлений о ценовых предложениях с использованием WebSocket-протокола, чтобы пользователь мог подписаться и наблюдать за ценами на выбранный товар.

Основные игроки, вовлекаемые в реализацию поиска товара, представлены на рис. 8.13.

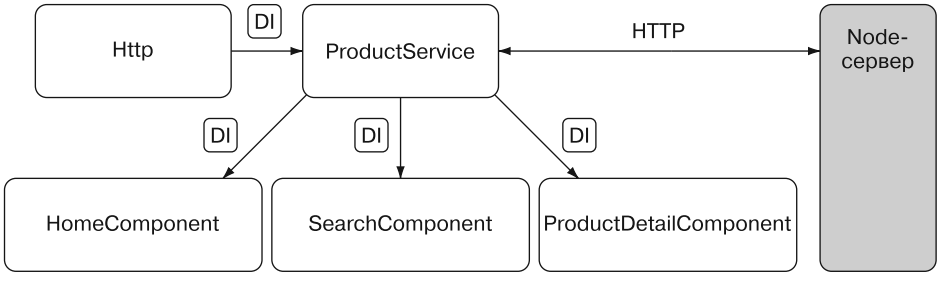


Рис. 8.13. Реализация поиска товара

Обозначение DI на рисунке означает *внедрение зависимости* (dependency injection). Angular внедряет `Http`-объект в `ProductService`, который, в свою очередь, внедряет три компонента: `HomeComponent`, `SearchComponent` и `ProductDetailComponent`. Объект `ProductService` отвечает за весь обмен данными с сервером.

ПРИМЕЧАНИЕ

В этом проекте используется Node-сервер, но вы можете воспользоваться любой технологией, поддерживающей протоколы HTTP и WebSocket, например Java, .NET, Python, Ruby и т. д.

Как уже упоминалось, мы дадим краткое пояснение относительно изменений в коде, внесенных в различные сценарии, но вам придется самостоятельно выполнить подробный пересмотр кода аукциона. В версии аукциона, предлагаемой в этой главе, раздел `scripts` выглядит следующим образом:

```
"scripts": {
    "tsc": "tsc",
    "start": "node build/auction.js",
    "dev": "nodemon build/auction.js"
}
```

Ввод команды `npm start` приведет к запуску вашего Node-сервера, загружающего сценарий `auction.js`. В этом проекте в файле `tsconfig.json` каталог `build` указывается в качестве выходного для TypeScript-компилятора. При вводе команды `npm run tsc` во время нахождения в корневом каталоге проекта создаются два файла, `auction.js` и `model.js`. Если в вашем распоряжении имеется версия компилятора TypeScript 2.0 или выше, установленная глобально, то можно просто запустить команду `tsc`.

В исходном TypeScript-файле `auction.ts` содержится код, реализующий серверы HTTP и WebSocket, а в файле `model.ts` — данные, которые теперь размещаются на сервере. Ввод команды `npm run dev` приведет к запуску вашего Node-сервера в режиме живой перезагрузки.

8.5.1. Реализация поиска товара с использованием HTTP

Главная страница аукциона имеет в левой части форму поиска Search (Поиск); пользователь может ввести критерий поиска, нажать кнопку с таким же именем и получить данные о соответствующем товаре с сервера. Как показано на рис. 8.13, класс `ProductService` несет ответственность за весь обмен данными с сервером по протоколу HTTP, включая начальную загрузку сведений о товарах или поиск товаров по конкретным критериям.

Помещение сведений о товарах и обзора на сервер

До сих пор данные, касающиеся товаров, и обзоры были жестко заданы в коде на клиентской стороне в классе `ProductService`; при запуске приложения оно показывало все жестко заданные товары в компоненте `HomeComponent`. При щелчке пользователя на товаре маршрутизатор выполнял переход к компоненту `ProductDetailComponent`, который показывал сведения о товарах и обзоры, также жестко заданные в `ProductService`.

Теперь нужно, чтобы сведения о товарах и обзоры размещались на сервере. Код, который будет запущен как Node-приложение (веб-сервер), содержится в файлах `server/auction.ts` и `server/model.ts`. В файле `auction.ts` реализованы функциональные свойства HTTP и WebSocket, а в файле `model.ts` объявляются классы `Product` и `Review`, а также массивы с данными `products` и `reviews`. Кроме того, эти массивы были удалены из файла `client/app/services/product-service.ts`.

ПРИМЕЧАНИЕ

Класс Product имеет новое свойство categories, которое будет использоваться в SearchComponent.

Класс ProductService

Этот класс получит внедренный `Http`-объект, и большинство методов данного класса станут возвращать наблюдаемые потоки, созданные HTTP-запросами. Новая версия метода `getProducts()` показана в следующем фрагменте кода:

```
getProducts(): Observable<Product[]> {
  return this.http.get('/products')
    .map(response => response.json());
}
```

Следует напомнить, что предыдущий метод не выдавал HTTP-запрос `GET`, пока какой-либо объект не подписывался на метод `getProducts()` или же пока шаблон компонента не использовал в отношении возвращаемых этим методом данных канал `AsyncPipe` (пример можно найти в компоненте `HomeComponent`).

Похоже выглядит и метод `getProductById()`:

```
getProductById(productId: number): Observable<Product> {
  return this.http.get(`/products/${productId}`)
    .map(response => response.json());
}
```

Метод `getReviewsForProduct()` также возвращает значение типа `Observable`:

```
getReviewsForProduct(productId: number): Observable<Review[]> {
  return this.http
     .get(`/products/${productId}/reviews`)
     .map(response => response.json())
     .map(reviews => reviews.map(
       (r: any) => new Review(r.id, r.productId, new Date(r.timestamp),
       ➥ r.user, r.rating, r.comment)));
}
```

Новый метод `ProductService.search()` используется, когда пользователь нажимает кнопку Search (Поиск), принадлежащую компоненту `SearchComponent`:

```
search(params: ProductSearchParams): Observable<Product[]> {
    return this.http
      .get('/products', {search: encodeParams(params)})
      .map(response => response.json());
}
```

Предыдущий метод `Http.get()` применяет второй аргумент, являющийся объектом со свойством `search` для хранения параметров строки запроса. Нетрудно заметить, что ранее в интерфейсе `RequestOptionsArgs` свойство `search` могло хранить либо строку, либо экземпляр класса `URLSearchParams`.

Далее показан код метода `ProductService.encodeParams()`, превращающего объект JavaScript в экземпляр класса `URLSearchParams`:

```
function encodeParams(params: any): URLSearchParams {
  return Object.keys(params)
    .filter(key => params[key])
    .reduce((accum: URLSearchParams, key: string) => {
      accum.append(key, params[key]);
      return accum;
    }, new URLSearchParams());
}
```

Новый метод `ProductService.getAllCategories()` используется для заполнения раскрывающегося списка Categories (Категории) в компоненте `SearchComponent`:

```
getAllCategories(): string[] {
  return ['Books', 'Electronics', 'Hardware'];
}
```

В классе `ProductService` также определяется новая переменная `searchEvent` типа `EventEmitter`. Ее предназначение будет объяснено ниже.

Предоставление результатов поиска компоненту HomeComponent

Изначально компонент `HomeComponent` отображает на экране все товары путем вызова метода `ProductService.getProducts()`. Но когда пользователь выполняет поиск по какому-либо критерию, вам нужен запрос к серверу, который может возвратить поднабор товаров или пустой набор данных, если критерию поиска не отвечает ни один из товаров.

Компонент `SearchComponent` получает результат, который должен быть передан компоненту `HomeComponent`. Если оба этих элемента были дочерними для общего родителя (например, `AppComponent`), то родительский компонент может использоваться как посредник (см. главу 6) и вводить-выводить в качестве данных параметры дочерних компонентов. Но `HomeComponent` добавляется в `AppComponent` маршрутизатором в динамическом режиме, а текущая версия Angular не поддерживает кросс-маршрутный ввод-вывод параметров. Вам нужен другой посредник; таковым может выступить объект `ProductService`, поскольку внедрен и в `SearchComponent`, и в `HomeComponent`.

В классе `ProductService` имеется переменная `searchEvent`, объявляемая следующим образом:

```
searchEvent: EventEmitter = new EventEmitter();
```

Компонент `SearchComponent` использует данную переменную для выдачи события `searchEvent`, которое в качестве полезной нагрузки переносит объект с параметрами поиска. Как показано на рис. 8.14, на это событие подписывается компонент `HomeComponent`.

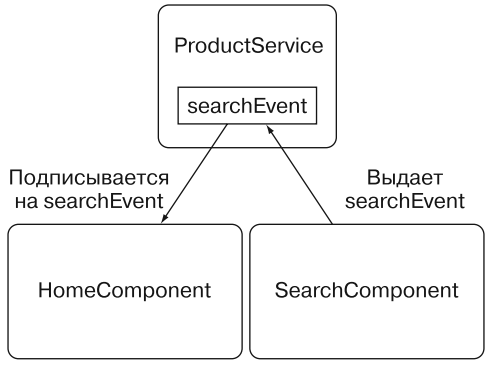


Рис. 8.14. Обмен данными между компонентами с помощью событий

Компонент `SearchComponent` является формой, и когда пользователь нажимает кнопку Search (Поиск), этот компонент должен уведомить мир о том, какие параметры поиска были введены. `ProductService` делает это, выдавая событие с параметрами поиска:

```
onSearch() {
  if (this.formModel.valid) {
    this.productService.searchEvent.emit(this.formModel.value);
  }
}
```

Компонент `HomeComponent` подписан на событие `searchEvent`, которое может прибыть из компонента `SearchComponent` с полезной нагрузкой в виде параметров поиска. Как только это произойдет, будет вызван метод `ProductService.search()`:

```
this.productService.searchEvent
  .subscribe(
```

```
    params => this.products = this.productService.search(params),
    console.error.bind(console),
    () => console.log('DONE')
);
```

Ограничения поиска

В нашем поисковом решении предполагается, что при выполнении пользователем поиска товара компонент `HomeComponent` показан на экране. Но если пользователь перейдет к представлению Product Detail (Информация о продукте), то данный компонент будет удален из DOM-модели и отслеживателей события `searchEvent` не станет. Для примера в книге это не является существенным недостатком, и проще всего будет исправить ситуацию, отключив кнопку поиска в случае ухода пользователя с маршрута `Home`. Можно также внедрить объект `Router` в компонент `SearchComponent` и, когда пользователь нажимает кнопку Search (Поиск) при неактивном состоянии главного маршрута (`if (!router.isActive(url))`), выполнить программный переход путем вызова метода `router.navigate('home')`, который возвратит промис в виде `Promise`-объекта. Когда промис будет разрешен, вы можете выдать из него событие `searchEvent`.

Обработка поиска товара на сервере

Следующий фрагмент кода взят из файла `auction.ts`, в коде которого ведется обработка запроса на поиск товара, отправленного клиентом. Когда клиент попадает в конечную точку сервера со строковыми параметрами запроса, полученные параметры передаются в виде `req.query` функции `getProducts()`. Она вводит в действие последовательность фильтров (в соответствии с установками параметров) в отношении массива товаров, чтобы отфильтровать неподходящие товары:

```
app.get('/products', (req, res) => {
  res.json(getProducts(req.query));
});
...
function getProducts(params): Product[] {
  let result = products;
  if (params.title) {
    result = result.filter(
      p => p.title.toLowerCase().indexOf(params.title.toLowerCase()) !== -1);
  }
  if ( result.length > 0 && parseInt(params.price)) {
    result = result.filter(
      p => p.price <= parseInt(params.price));
  }
  if ( result.length > 0 && params.category) {
    result = result.filter(
      p => p.categories.indexOf(params.category.toLowerCase()) !== -1);
  }
  return result;
}
```

Тестирование функционирования поиска товаров

После краткого обзора кода реализации поиска товаров можно запустить Node-сервер, воспользовавшись командой `npm run dev`, и открыть в браузере адрес localhost:8000. После загрузки приложения-аукциона введите свой критерий поиска в форму в левой части окна и посмотрите, как компонент `HomeComponent` заново отобразит свой дочерний элемент (`ProductItemComponent`), отвечающий критериям поиска.

8.5.2. Распространение по сети ценовых предложений аукциона с использованием веб-сокетов

При проведении реальных аукционов ценовые предложения могут делаться сразу несколькими пользователями. Когда сервер получает такое предложение от пользователя, сервер ценовых предложений должен распространить по сети самое последнее из них среди всех пользователей, заинтересованных в подобных уведомлениях (среди подписавшихся на уведомления). Процесс выдачи ценовых предложений будет имитироваться путем генерации случайных цен от случайных пользователей.

Когда пользователи открывают представление Product Details (Информация о продукте), они должны иметь возможность подписаться на уведомление о ценовых предложениях на выбранный товар, сделанных другими пользователями. Эта функциональная возможность реализуется выдачей с серверной стороны через веб-сокеты. Представление Product Details (Информация о продукте) с кнопкой переключения Watch (???), запускающей и останавливающей текущие уведомления о ценовых предложениях, выдаваемых сервером с помощью сокета, показано на рис. 8.15. Далее мы кратко охарактеризуем изменения в приложении-аукционе, связанные с уведомлениями о ценовых предложениях.

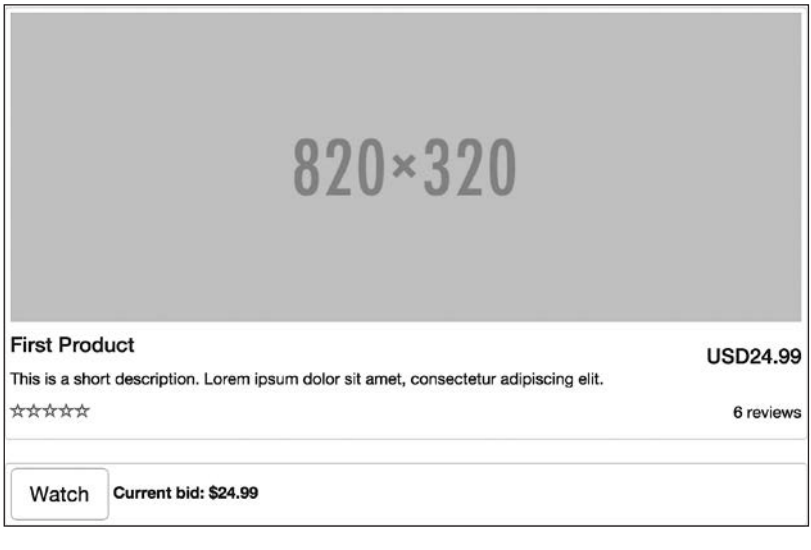


Рис. 8.15. Кнопка-переключатель для отслеживания ценовых предложений

Клиентская сторона

В каталоге `client/app/services` размещены два новых сервиса: `BidService` и `WebSocketService`. Последний является наблюдаемой оболочкой `Observable` для `WebSocket`-объекта. Он похож на тот, который создавался ранее в подразделе 8.4.2.

Сервис `BidService` получает внедренный сервис `WebSocketService`:

```
@Injectable()
export class BidService {
  constructor(private webSocket: WebSocketService) {}
  watchProduct(productId: number): Observable {
    let openSubscriber = Subscriber.create(
      () => this.webSocket.send({productId: productId}));
    return this.webSocket.createObservableSocket('ws://
     localhost:8000', openSubscriber)
      .map(message => JSON.parse(message));
  }
}
```

Сервис `BidService` внедрен в компонент `ProductDetailComponent`. Когда пользователь нажимает кнопку переключения Watch (???), метод `BidService.watchProduct()` отправляет идентификатор товара на сервер, показывая тем самым, что данный пользователь хочет запустить или остановить наблюдение за выбранным товаром:

```
toggleWatchProduct() {
  if (this.subscription) {
    this.subscription.unsubscribe();
    this.subscription = null;
    this.isWatching = false;
  } else {
    this.isWatching = true;
    this.subscription = this.bidService.watchProduct(this.product.id)
      .subscribe(
        products => this.currentBid = products.find((p: any) => p.productId
        ➥ === this.product.id).bid,
        error => console.log(error));
  }
}
```

У шаблона компонента `ProductDetailComponent` имеется кнопка переключения Watch (???), и самые последние ценовые предложения, получаемые от сервера, отображаются в виде HTML-надписи:

```
<button class="btn btn-default btn-lg"
          [ngClass]="{active: isWatching}"
          (click)="toggleWatchProduct()"
          role="button">
  {{ isWatching ? 'Stop watching' : 'Watch' }}
</button>
<label>Current bid: {{ currentBid | currency }}</label>
```

Еще есть небольшой новый сценарий `client/app/services/services.ts`, в котором объявляются все инструкции импортирования и массив сервисов, используемых для внедрения зависимостей:

```
import {BidService} from './bid-service';
import {ProductService} from './product-service';
import {WebSocketService} from './websocket-service';
export const ONLINE_AUCTION_SERVICES = [
  BidService,
  ProductService,
  WebSocketService
];
```

Поставщики, объявленные в константе `ONLINE_AUCTION_SERVICES`, используются в файле `main.ts`, загружающем ту часть аукциона, которая относится к Angular:

```
@NgModule({
...
  providers:[ProductService,
             ONLINE_AUCTION_SERVICES,
             {provide: LocationStrategy, useClass: HashLocationStrategy}],
  bootstrap:[ ApplicationComponent ]
})
```

Серверная сторона

Сценарий `server/auction.ts` включает код, обслуживающий подписавшихся клиентов и генерирующий случайные ценовые предложения. Каждое сгенерированное ценовое предложение должно быть на пять долларов выше предыдущего предложения. Как только будет сгенерировано новое ценовое предложение, оно тут же распространяется среди всех подписавшихся клиентов.

Обслуживанием запросов на уведомления о ценовых предложениях и распространением ценовых предложений среди всех подписавшихся клиентов занимается следующий код из файла `server/auction.ts`:

Сохраняет ссылки на подписки о ценовых предложениях в отображении Map, где ключом служит ссылка на WebSocket-подключение, представляющее пользователя, а значением — массив идентификаторов товаров, для которых клиент желает получать уведомления о предложениях цены

```
const wsServer: WsServer = new WsServer({server: httpServer});

wsServer.on('connection', ws => {
  ws.on('message', message => {
    let subscriptionRequest = JSON.parse(message);
    subscribeToProductBids(ws, subscriptionRequest.productId);
  });
});


const subscriptions = new Map<any, number[]>();
```

Создает WebSocket-сервер, отслеживающий тот же самый порт, что и HTTP-сервер

```
function subscribeToProductBids(client, productId: number): void {
  let products = subscriptions.get(client) || [];
  subscriptions.set(client, [...products, productId]);
}
```

Ищет существующие подписки на товар для подключившегося клиента и добавляет новый идентификатор товара к массиву подписок

```
setInterval(() => {
  generateNewBids();
  broadcastNewBidsToSubscribers();
}, 2000);
```

Генерирует каждые две секунды новые предложения по цене и распространяет их среди всех клиентов, подписавшихся на уведомления, касающиеся конкретных товаров

```
const currentBids = new Map<number, number>();
function generateNewBids() {
  getProducts().forEach(p => {
    const currentBid = currentBids.get(p.id) || p.price;
    const newBid = random(currentBid, currentBid + 5); //
      Max bid increase is $5
    currentBids.set(p.id, newBid);
  });
}

function broadcastNewBidsToSubscribers() {

  subscriptions.forEach((products: number[], ws: WebSocket) => {
    if (ws.readyState === 1) { // 1 - READY_STATE_OPEN
      let newBids = products.map(pid => ({
        productId: pid
        bid: currentBids.get(pid)
      }));
      ws.send(JSON.stringify(newBids));
    } else {
      subscriptions.delete(ws);
    }
  });
}
```

Отправляет текущие предложения цены для товаров, на которые имеются подписки, каждому подключенному клиенту

Здесь вам следует протестировать значение свойства `readyState` WebSocket-объекта, чтобы убедиться в активности подключения клиента. Например, если пользователь закрыл окно аукциона, то надобность в отправке уведомлений о ценовых предложениях отпадает, поэтому данное сокет-подключение из отображения подписок удаляется.

ПРИМЕЧАНИЕ

Обратите внимание на использование в методе subscribeToProductBids() оператора распространения (...). Он применяется для копирования существующего массива идентификаторов товаров и добавления нового идентификатора.

Мы рассмотрели код аукциона, относящийся к веб-сокету, а весь остальной код предлагаем изучить самостоятельно. Для тестирования работы по уведомлениям

о ценовых предложениях нужно запустить приложение, щелкнуть на названии товара и в представлении сведений о товаре щелкнуть на кнопке Watch (???). Вы увидите новые ценовые предложения для этого товара, выданные сервером. Откройте приложение-аукцион в более чем одном браузере, чтобы протестировать, должным ли образом включаются и выключаются уведомления в каждом браузере.

8.6. Резюме

Основной темой данной главы было предоставление возможности клиент-серверного взаимодействия, являющегося основанием для существования веб-сред. Angular в сочетании с библиотекой расширений RxJS предлагает унифицированный подход использования данных, получаемых с сервера: клиентский код подписывается на поток данных, поступающий с сервера, независимо от характера взаимодействия, основанного либо на протоколе HTTP, либо на WebSocket. Модель программирования изменилась: вместо запроса данных как в приложениях стиля Ajax, Angular задействует данные, *выдаваемые* наблюдаемыми потоками.

Вот основные выводы этой главы.

- Angular поставляется с `Http`-объектом, поддерживающим обмен данными с веб-сервером по протоколу HTTP.
- Поставщики для HTTP-сервисов находятся в модуле `HttpModule`. Если в вашем приложении используется протокол HTTP, то не забудьте включить его в декоратор `@NgModule`.
- Открытые методы `HttpObject` возвращают объект типа `Observable`, но только когда клиент подпишется на него, выполняется запрос к серверу.
- Протокол WebSocket эффективнее и лаконичнее протокола HTTP. Он двунаправленный, и обмен данными может инициироваться как клиентом, так и сервером.
- Создание веб-сервера с помощью NodeJS и Express представляется относительно несложной задачей, но клиент Angular может вести обмен данными с веб-серверами, реализованными с использованием и других технологий.

9 Модульное тестирование Angular-приложений

В этой главе:

- основы модульного тестирования с применением среды Jasmine;
- базовые средства, получаемые из библиотеки тестирования Angular;
- тестирование основных исполнителей Angular-приложения: сервисов, компонентов и маршрутизатора;
- запуск модульных тестов на браузерах с использованием средства для запуска тестов Karma;
- реализация модульного тестирования на примере онлайн-аукциона.

Чтобы убедиться в отсутствии ошибок в программном средстве, его нужно протестировать. Даже если в нем сегодня нет никаких ошибок, они могут появиться завтра, после внесения изменений в существующий код или введения нового кода. Пусть код в отдельно взятом модуле и не изменялся, но он может заработать неправильно в силу изменений в каком-нибудь другом модуле. Код вашего приложения должен регулярно подвергаться тестированию, и этот процесс нужно автоматизировать. Вам следует подготовить тестовые сценарии и приступить к их запуску в вашем разработочном цикле как можно раньше.

Существует два основных типа тестирования клиентской части программы веб-приложения.

- *Модульное (блочное) тестирование*, которое доказывает, что небольшие блоки кода (например, компоненты или функции) воспринимают ожидаемые входные данные и возвращают ожидаемый результат. Данный вид тестирования касается проверки изолированных частей кода, главным образом открытых интерфейсов. Именно это мы и будем рассматривать в настоящей главе.
- *Сквозное тестирование*, доказывающее работоспособность всего приложения в рамках ожиданий конечных пользователей и правильную организацию взаимодействия всех блоком друг с другом. Для сквозного тестирования приложений Angular можно воспользоваться библиотекой Protractor (см. http://www.protractortest.org/#/).

ПРИМЕЧАНИЕ

Нагрузочное тестирование, или стресс-тест, показывает, сколько пользователей могут одновременно работать с веб-приложением при сохранении им ожидаемого времени отклика. Средства нагрузочного тестирования в основном касаются проверки серверных сторон веб-приложений.

Модульное тестирование предназначено для проверки логики функционирования отдельных модулей кода, и, как правило, такие тесты запускаются намного чаще сквозных. Последние могут имитировать действия пользователей (например, нажатия кнопок) и проверять поведение вашего приложения. Запускать сценарии модульного тестирования в ходе сквозного нельзя.

Данная глава посвящена модульному тестированию Angular-приложений. Для реализации и запуска модульных тестов существует целый ряд специально созданных сред, и наш выбор пал на Jasmine. Фактически это не только наш выбор; на момент написания этих строк имеющаяся в Angular библиотека тестирования работала в области модульного тестирования только с Jasmine. Соответствующее описание приведено в разделе Jasmine Testing 101 документации по Angular (https://angular.io/guide/testing#!#jasmine-101).

Сначала мы рассмотрим основы модульного тестирования с применением среды Jasmine, а ближе к концу главы вы создадите и запустите сценарии для модульного тестирования компонентов онлайн-аукциона. Мы предоставим краткий обзор среды Jasmine, чтобы поспособствовать быстрому переходу к созданию модульных тестов; все остальные подробности можно будет найти в документации по Jasmine (http://jasmine.github.io). Для запуска тестов будет использоваться специальное средство под названием Karma (https://karma-runner.github.io/1.0/index.html), которое является независимой утилитой командной строки, способной запускать тесты, написанные в различных средах тестирования.

9.1. Знакомство с Jasmine

Среда Jasmine позволяет реализовать процесс разработки через реализацию поведения (behavior-driven development (BDD)), предполагающий, что тесты любого блока программного средства должны формулироваться в понятиях желаемого поведения блока. При использовании BDD для описания ваших замыслов относительно предназначения кода применяются обычные языковые конструкции. Спецификации модульных тестов пишутся в виде коротких предложений, например, ApplicationComponent is successfully instantiated (Экземпляр ApplicationComponent успешно создан) или StarsComponent emits the rating change event (StarsComponent выдает событие изменения рейтинга).

Поскольку назначения тестов воспринимаются довольно легко, они могут послужить в качестве документации вашей программы. Когда другим разработчикам понадобится ознакомиться с вашим кодом, для выяснения ваших намерений они могут приступить к чтению кода модульных тестов. Использование обычного языка для описания тестов дает еще одно преимущество: простоту понимания результатов тестирования, в чем можно убедиться, посмотрев на рис. 9.1.

Рис. 9.1. Запуск тестов с применением исполнителя тестов среды Jasmine

По терминологии Jasmine тест называется *спецификацией* (spec), а комбинация нескольких спецификаций — *набором* (suite). Тестовый набор определяется функцией `describe()`: именно здесь дается описание тому, что проверяется. Каждая тестовая спецификация программируется как функция `it()`, в которой определяется ожидаемое поведение анализируемого кода и порядок его тестирования. Рассмотрим пример:

```
describe('MyCalculator', () => {
  it('should know how to multiply', () => {
    // Сюда помещается код, тестирующий умножение
  });

  it('should not divide by zero', () => {
    // Сюда помещается код, тестирующий деление на нуль
  });
});
```

В средах тестирования используется такое понятие, как *утверждение* (assertion), являющееся способом опроса, во что вычисляется выражение — в `true` или в `false`. Если утверждение имеет значение `false`, то среда выдает ошибку. В Jasmine утверждения определяются с помощью функции `expect()`, за которой следуют *сопоставления* (matchers): `toBe()`, `toEqual()` и т. д. Это похоже на написание предложения. I expect 2+2 to equal 4 (Я ожидаю, что 2 + 2 равно 4) выглядит следующим образом:

```
expect(2 + 2).toEqual(4);
```

Сопоставления реализуют булевы сравнения фактических и ожидаемых значений.

При возвращении сопоставлением значения `true` спецификация считается пройденной. Если ожидается, что у результата теста нет конкретного значения, то нужно перед сопоставлением добавить ключевое слово `not`:

```
expect(2 + 2).not.toEqual(5);
```

Полный список сопоставлений можно найти на ресурсе GitHub, на странице Джейми Мейсона (Jamie Mason), которая называется Jasmine-Matchers: https://github.com/JamieMason/Jasmine-Matchers.

Мы дали нашим тестовым наборам такие же имена, как и у тестируемых файлов, добавив к ним суффикс `.spec`, что является стандартным приемом; например, в `application.spec.ts` содержится тестовый сценарий для `application.ts`. Следующий тестовый набор взят из файла `application.spec.ts`; он тестирует создание экземпляра `ApplicationComponent`:

```
import AppComponent from './app';
describe('AppComponent', () => {
  it('is successfully instantiated', () => {
    const app = new AppComponent();
    expect(app instanceof AppComponent).toEqual(true);
  });
});
```

В этом тестовом наборе содержится всего один тест. Если извлечь тексты из `describe()` и `it()` и объединить их, то получится предложение, разъясняющее, что именно здесь тестируется: ApplicationComponent is successfully instantiated (Экземпляр ApplicationComponent успешно создан).

ПРИМЕЧАНИЕ

Если другим разработчикам понадобится узнать, что тестируется вашей спецификацией, то они могут прочитать тексты в describe() и в it(). Каждый тест должен давать свое собственное описание и служить в качестве программной документации.

В предыдущем коде создается экземпляр `AppComponent` и ожидается, что выражение `app instanceof AppComponent` будет вычислено в `true`. Присутствие инструкции `import` может навести вас на мысль о нахождении сценария тестирования в том же самом каталоге, в котором помещается и `AppComponent`.

Где следует хранить файлы с тестами

Среда Jasmine используется для модульного тестирования приложений на JavaScript, написанных в различных средах или на чистом JavaScript. Одним из подходов к хранению файлов с тестами является создание отдельного каталога `test` для хранения в нем исключительно файлов со сценариями тестов, чтобы они не смешивались с кодом приложения.

В Angular-приложениях мы отдали предпочтение хранению каждого тестового сценария в том же самом каталоге, где находится тестируемый компонент или сервис. Это удобно по двум причинам.

- Все файлы, имеющие отношение к компоненту, хранятся в одном каталоге. Обычно каталог создается для хранения принадлежащих компоненту файлов с расширениями `.ts`, `.html` и `.css`; добавление файла с расширением `.spec` не станет захламлять содержимое каталога.
- Не нужно изменять конфигурацию загрузчика SystemJS, который уже знает, где находятся файлы приложения. Он будет загружать тесты из того же самого места.

Если нужно, чтобы перед каждым тестом выполнялся какой-нибудь код (например, для подготовки тестовых зависимостей), то вы можете указать его в настроечных функциях `beforeAll()` и `beforeEach()`, которые будут запущены соответственно перед набором или перед каждой спецификацией. При необходимости выполнить какой-нибудь код сразу же после завершения набора или каждой спецификации следует воспользоваться демонтирующими функциями `afterAll()` и `afterEach()`.

СОВЕТ

Если у спецификации несколько тестов it() и нужно, чтобы средство запуска тестов пропустило некоторые из них, то измените их название с it() на xit().

9.1.1. Что именно тестировать

Теперь, когда стало понятно, как тестировать, остается спросить, что именно нужно проверять. В Angular-приложениях, написанных на TypeScript, можно тестировать функции, классы и компоненты.

- *Тестирование функций* — предположим, имеется функция, переводящая символы переданной строки в верхний регистр. Для нее можно написать сразу несколько тестов — для тех случаев, когда аргументом является `null`, пустая строка, неопределенное значение, слово с символами в нижнем регистре, слово с символами в верхнем регистре, слово с символами в смешанном регистре, число и т. д.
- *Тестирование классов* — если имеется класс, содержащий несколько методов (наподобие класса `ProductService`), то можно написать тестовый набор, включающий все тесты, необходимые для того, чтобы убедиться в правильном функционировании каждого из методов класса.
- *Тестирование компонентов* — можно проверять открытый API ваших сервисов или компонентов. Помимо тестирования их на корректность работы, вам будут показаны примеры кода, использующие общедоступные свойства или методы.

9.1.2. Порядок установки Jasmine

Получить Jasmine можно путем загрузки автономного дистрибутива этой среды, но вы установите ее с помощью npm, как делалось для всех других пакетов в данной книге. В хранилище npm имеется несколько связанных с Jasmine пакетов, но вам нужен только `jasmine-core`. Откройте в корневом каталоге своего проекта окно команд и запустите следующую команду:

```
npm install jasmine-core --save-dev
```

Чтобы компилятор TypeScript разбирался в типах среды Jasmine, запустите следующую команду для установки имеющегося у Jasmine файла определения типов:

```
npm i @types/jasmine --save-dev
```

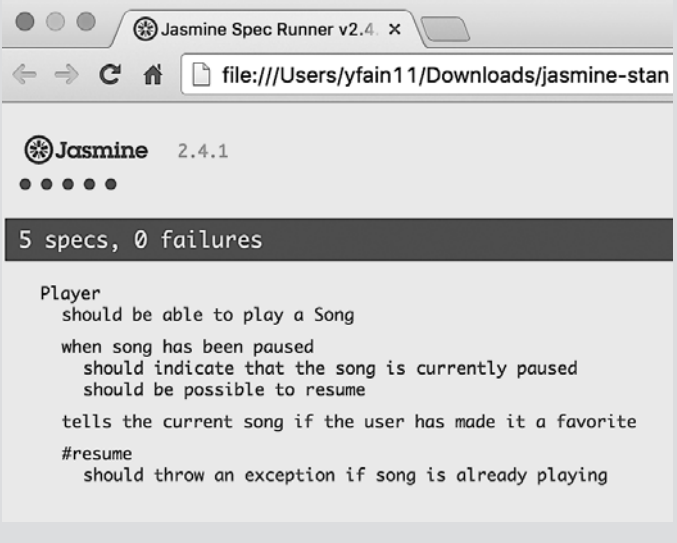
После написания тестов вам понадобится приложение для их запуска. Jasmine поставляется с двумя пусковыми средствами, одно из которых предназначено для работы с командной строкой (см. npm-пакет jasmine), а другое основано на применении кода HTML. Начнем с пускового средства на основе HTML, но для запуска тестов из командной строки будет использоваться другая программа — Karma.

Хотя Jasmine поставляется с заранее сконфигурированным средством запуска на основе HTML в виде образцового приложения, для тестирования своего собственного кода вам нужно будет создать HTML-файл. В него должны быть включены следующие сценарные теги, загружающие Jasmine:

```
<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>
```

Использование автономного дистрибутива Jasmine

Если вам хочется поскорее увидеть запущенными тесты Jasmine, то загрузите zip-файл с автономной версией Jasmine с https://github.com/jasmine/jasmine/releases. Распакуйте его и откройте в своем браузере файл `SpecRunner.html`. Вы должны увидеть там следующее окно.



Тестирование образцового приложения, поставляемого вместе с Jasmine

Нужно также добавить все востребованные Angular зависимости, как делалось в каждом файле `index.html` во всех примерах кода в данной книге, плюс библиотеку тестирования Angular. Продолжим использовать загрузчик SystemJS, но на этот раз загрузим код модульных тестов (файлы с расширением `.spec files`), которые загрузят код приложения через операторы `import`.

В этой главе мы опишем процесс создания модульных тестов. Запускать их будем вручную, сначала с помощью средств запуска на основе HTML. Затем покажем порядок использования программы Karma, которая может запускать тесты командной строки, сообщающие в различных браузерах о возможных ошибках. В главе 10 она будет встроена в процесс сборки приложения, чтобы модульные тесты запускались автоматически как часть сборки.

9.2. Средства, предоставляемые библиотекой тестирования Angular

Angular поставляется с библиотекой тестирования, включающей оболочки для имеющихся в Jasmine функций `describe()`, `it()` и `xit()`, а также добавляющей такие функции, как `beforeEach()`, `async()`, `fakeAsync()` и др.

Поскольку в ходе выполнения тестов приложение не настраивается и не загружается, Angular предоставляет вспомогательный класс `TestBed`, который позволяет объявлять модули, компоненты, поставщики и т. д. Этот класс включает такие функции, как `configureTestingModule()`, `createComponent()`, `inject()` и др. Например, синтаксис для настройки модуля тестирования похож на настройку аннотации `@NgModule`:

```
beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [ ReactiveFormsModule, RouterTestingModule,
                 RouterTestingModule.withRoutes(routes)],
      declarations: [AppComponent, HomeComponent, WeatherComponent],
      providers: [{provide: WeatherService, useValue: {} }
      ]
  })
});
```

Функция `beforeEach()` используется в тестовых наборах в фазе настройки. Она позволяет указать требуемые модули, компоненты и поставщики, которые могут понадобиться для каждого теста.

Функция `inject()` создает средство внедрения и вводит конкретные объекты в тесты, в соответствии с поставщиками приложения, настроенными для Angular DI:

```
inject([Router, Location], (router: Router, location: Location) => {
  // Выполнение каких-либо действий
})
```

Функция `async()` запускается в Zone и может использоваться с асинхронными сервисами. Эта функция не заканчивает тест до тех пор, пока не завершатся все его асинхронные операции или не истечет указанное время ожидания:

```
it(' does something', async(inject([AClass], object => {
  myPromise.then(() => { expect(true).toEqual(true); });
}), 3000));
```

Функция `fakeAsync()` позволяет ускорить тестирование асинхронных сервисов путем имитации течения времени:

```
it('...', fakeAsync(() => {
  // Выполнение каких-либо действий

  tick(1000);
  expect(...);
}));
```

Имитирует асинхронное течение времени сроком одна секунда

В библиотеке тестирования Angular имеется интерфейс `NgMatchers`, который включает следующие сопоставления:

- `toBePromise()` — ожидает, что значением будет `Promise`-объект;
- `toBeAnInstanceOf()` — ожидает, что значением будет экземпляр класса;
- `toHaveText()` — ожидает, что у элемента имеется в точности заданный текст;
- `toHaveCssClass()` — ожидает, что у элемента имеется заданный класс CSS;
- `toHaveCssStyle()` — ожидает, что у элемента имеются заданные стили CSS;
- `toImplement()` — ожидает, что в классе реализован интерфейс заданного класса;
- `toContainError()` — ожидает, что в исключении содержится заданный текст ошибки;
- `toThrowErrorWith()` — ожидает, что функция при выполнении выдаст ошибку с заданным текстом ошибки.

Документация по имеющемуся в Angular API тестирования для TypeScript может быть найдена на https://angular.io/guide/testing. Порядок тестирования сервисов, маршрутизаторов, источников событий и компонентов будет показан в этой главе чуть позже, но сначала рассмотрим некоторые основы.

9.2.1. Тестирование сервисов

Обычно Angular-сервисы внедряются в компоненты; для настройки средств внедрения нужно определить поставщики для блока `it()`. Angular предлагает настроечный метод `beforeEach()`, запускаемый перед каждым запуском `it()`. Чтобы протестировать в сервисе синхронные функции, можно воспользоваться методом `inject()` и внедрить этот сервис в `it()`.

Реальным сервисам для завершения работы может понадобиться некоторое время, и это способно замедлить ваши тесты. Есть два способа их ускорения.

- Создание класса, реализующего сервис-имитатор, который сможет быстро возвратить жестко заданные данные за счет расширения класса настоящего сервиса. Например, можно создать сервис-имитатор для `WeatherService`, тут же

возвращающий данные без выполнения каких-либо запросов к удаленному серверу на возвращение фактических погодных данных:

```
class MockWeatherService implement WeatherService {
  getWeather() {
    return Observable.empty();
  }
}
```

- Применение функции `fakeAsync()`, которая автоматически идентифицирует асинхронные вызовы и заменяет паузы, функции обратного вызова и `Promise`-объекты функциями, выполняемыми без всяких промедлений. Функция `tick()` позволяет ускорить время, чтобы не нужно было ожидать истечения срока паузы. Примеры использования `fakeAsync()` будут показаны в данной главе чуть позже.

9.2.2. Тестирование навигации с помощью маршрутизатора

Для тестирования маршрутизатора сценарии спецификаторов могут вызывать такие методы маршрутизации, как `navigate()` и `navigateByUrl()`. Первый метод получает массив сконфигурированных маршрутов (команд), выстраивающих маршрут в виде аргумента, а второй метод получает строку, представляющую собой сегмент URL, по которому нужно выполнить переход.

В процессе использования метода `navigate()` указываются параметры сконфигурированного пути и маршрута, если таковые имеются. При правильной конфигурации маршрутизатора он должен обновить URL в адресной строке браузера.

В следующем фрагменте программного кода показано, как программным способом перейти к маршруту товара `product`, передать в качестве параметра маршрута `0` и убедиться в том, что после перехода веб-адрес (представленный объектом типа `Location`) имеет сегмент `/product/0`:

```
it('should be able to navigate to product details using commands API',
    fakeAsync(inject([Router, Location], (router: Router, location:
    ➥ Location) => {
      TestBed.createComponent(AppComponent);
      router.navigate(['/products', 0]);
      tick();
      expect(location.path()).toBe('/product/0');
  })
));
```

Когда маршрутизатору предоставляется массив значений, им вызывается *API команд*. Чтобы заработал предыдущий фрагмент кода, маршрутизатор с параметром `/products/:productId` должен быть сконфигурирован в соответствии с объяснениями, которые даны в главе 3.

Функция `it()` вызывает функцию обратного вызова, предоставляемую в качестве второго аргумента. Метод `fakeAsync()` инкапсулирует функцию, предоставленную в качестве аргумента (в предыдущем примере это `inject()`), и выполняет ее в Zone. Функция `tick()` позволяет вручную ускорить время и переносить на более ранний срок задачи в очереди микрозадач браузерного цикла событий. Иными

словами, можно имитировать время, занимаемое решением асинхронных задач, и выполнять асинхронный код в синхронном режиме, что упрощает и ускоряет проведение модульных тестов.

При использовании метода `TestBed.createComponent()` (рассматриваемого в следующем разделе) создается экземпляр компонента. В момент вызова метода маршрутизатора `navigate()` с помощью функции `tick()` ускоряются асинхронные задачи, выполняющие навигацию, и проверяется соответствие текущего местоположения ожидаемому.

Функция `navigateByUrl()` получает конкретный сегмент URL и должна правильно выстроить `Location.path`, представляющий клиентскую часть в адресной строке браузера. Вот то, что будет тестироваться:

```
router.navigateByUrl('/products');
...
expect(location.path()).toBe('/products');
```

Порядок использования функции `navigateByUrl()` будет показан в разделе 9.3.

При тестировании маршрутизатора можно воспользоваться `SpyLocation` — это имитатор поставщика `Location`. Он позволяет проводить тесты имитации событий местоположений. Например, можно подготовить конкретный веб-адрес и имитировать изменение хеш-части, работу кнопок браузера Back (Назад) и Forward (Вперед) и многое другое.

9.2.3. Тестирование компонентов

Компоненты представляют собой классы с шаблонами. Если в классе содержатся методы, реализующие логику приложения, то их можно протестировать, как и любые другие функции; но чаще всего будут тестироваться шаблоны. В частности, вы можете проявить интерес к тестированию правильной работы привязок и к проверке того, какие данные отображают шаблоны.

Angular предлагает метод `TestBed.createComponent()`, возвращающий объект `ComponentFixture`, который будет использоваться для работы с компонентами при их создании. Это приспособление (fixture) дает доступ как к компоненту, так и к собственным экземплярам HTML-элементов, позволяя присваивать значения свойствам компонентов, а также находить конкретные HTML-элементы в шаблоне компонента.

Кроме того, можно активизировать циклическое определение изменений путем вызова в отношении объекта-приспособления метода `detectChanges()`. После того как определение изменения обновило пользовательский интерфейс (UI), можно запустить функцию `expect()`, чтобы проверить выведенные значения. Эти действия показаны в следующем фрагменте кода с помощью компонента `ProductComponent`, имеющего свойство `product`, при условии его привязки к элементу шаблона `<h4>`:

```
let fixture = TestBed.createComponent(ProductDetailComponent);
let element = fixture.nativeElement;
let component = fixture.componentInstance;
component.product = {title: 'iPhone 7', price: 700};
fixture.detectChanges();
expect(element.querySelector('h4').innerHTML).toBe('iPhone 7');
```

А теперь создадим типовое приложение, в котором реализуем модульное тестирование компонента, маршрутизатора и сервиса.

9.3. Пример: тестирование приложения для составления прогнозов погоды

Попробуем протестировать компоненты и сервисы Angular, используя приложение, имеющее главную страницу с двумя ссылками: `Home` и `Weather`. Для перехода на страницу погоды `Weather` будет применяться маршрутизатор, являющийся реструктурированной версией приложения для прогноза погоды, созданного в главе 5 (`observable-events-http.ts`).

В главе 5 большой фрагмент кода был помещен в конструктор `AppComponent`, что усложняет тестирование, поскольку не дает возможности вызвать код конструктора после создания объекта. Теперь `WeatherComponent` получит внедрение `WeatherService`, и этот сервис станет использовать удаленный сервер из главы 5 для получения информации о погоде. На рис. 9.2 показано, как выглядит окно при запуске данного приложения после перехода по маршруту `Weather` и набора в поле ввода строки `New York`.

Структура этого проекта показана на рис. 9.3 (см. каталог `test_weather`). Обратите внимание на файлы с расширением `.spec`, в которых содержится код для модульного тестирования компонентов и сервиса погоды.

Рис. 9.2. Проверка погодного компонента в проекте test_samples

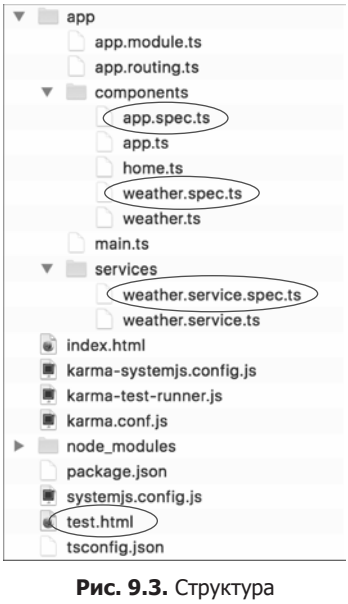


Рис. 9.3. Структура проекта test_samples

Для запуска этих тестов создайте следующий файл `test.html` (листинг 9.1), загружающий все файлы `spec.ts`, обведенные на рис. 9.3.

Листинг 9.1. Содержимое файла test.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Testing the Weather Application</title>
  <base href="/">

  <!-- TypeScript in-browser compiler -->
  <script src="node_modules/typescript/lib/typescript.js"></script>

  <!-- Polyfills -->
  <script src="node_modules/reflect-metadata/Reflect.js"></script>
  <!-- Jasmine -->                          Загрузка файлов Jasmine.
  <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>

  <!-- Zone.js -->
  <script src="node_modules/zone.js/dist/zone.js"></script>
  <script src="node_modules/zone.js/dist/proxy.js"></script>
  <script src="node_modules/zone.js/dist/sync-test.js"></script>
  <script src="node_modules/zone.js/dist/jasmine-patch.js"></script>
  <script src="node_modules/zone.js/dist/async-test.js"></script>
  <script src="node_modules/zone.js/dist/fake-async-test.js"></script>
  <script src="node_modules/zone.js/dist/long-stack-trace-zone.js"></script>

  <!-- SystemJS -->
  <script src="node_modules/systemjs/dist/system.src.js"></script>
  <script src="systemjs.config.js"></script>
</head>
<body>
<script>                                   Объявление массива с именами
  var SPEC_MODULES = [                     загружаемых спецификаций
    'app/components/app.spec',
    'app/components/weather.spec',
    'app/services/weather.service.spec'
  ];
                                           Загрузка двух Angular-модулей,
  Promise.all([                            требуемых для тестирования
    System.import('@angular/core/testing'),
    System.import('@angular/platform-browser-dynamic/testing')
  ])
  .then(function (modules) {               Когда тестовые модули загружены,
    var testing = modules[0];              сохранение их ссылок в переменных
    var browser = modules[1];

     testing.TestBed.initTestEnvironment(  Инициализация тестовой
         browser.BrowserDynamicTestingModule,  среды. Если в приложении
         browser.platformBrowserDynamicTesting());  сконфигурирован модуль
                                           AppModule, то в тестах он
                                           заменяется модулем
                                           BrowserDynamicTestingModule
```

```
    // Load all the spec files.
    return Promise.all(SPEC_MODULES.map(function (module) {
      return System.import(module);
    }));
  })
  .then(window.onload)
  .catch(console.error.bind(console));
</script>
</body>
</html>
```

После загрузки среды и спецификаций инициирование обработчика событий для события загрузки, чтобы Jasmine запустил тесты

9.3.1. Настройка SystemJS

Чтобы воспользоваться средством запуска тестов на основе HTML, нужно добавить модули тестирования Angular в настройки вашего загрузчика модулей SystemJS. Фрагмент файла `systemjs.config.js`, поставляемого вместе с проектом, выглядит следующим образом (листинг 9.2).

Листинг 9.2. Фрагмент файла systemjs.config.js

```
'@angular/common/testing'                    : 'ng:common/bundles/
➥ common-testing.umd.js',
    '@angular/compiler/testing'                : 'ng:compiler/bundles/
    ➥ compiler-testing.umd.js',
    '@angular/core/testing'                    : 'ng:core/bundles/
    ➥ core-testing.umd.js',
    '@angular/router/testing'                    : 'ng:router/bundles/
    ➥ router-testing.umd.js',
    '@angular/http/testing'                    : 'ng:http/bundles/
    ➥ http-testing.umd.js',
    '@angular/platform-browser/testing'        : 'ng:platform-browser/
    ➥ bundles/platform-browser-testing.umd.js',
    '@angular/platform-browser-dynamic/testing':
'ng:platform-browser-dynamic/bundles/platform-browser-dynamic-testing.umd.js',
  },
  paths: {
    'ng:': 'node_modules/@angular/'
  },
```

9.3.2. Тестирование маршрутизатора приложения для составления прогнозов погоды

Маршрутизатор для этого приложения конфигурируется в файле `app.routing.ts` (листинг 9.3).

Листинг 9.3. Содержимое файла app.routing.ts

```
import { Routes, RouterModule } from '@angular/router';
import { HomeComponent } from './components/home';
import { WeatherComponent } from './components/weather';
export const routes: Routes = [
```

```
  { path: '',        component: HomeComponent },
  { path: 'weather', component: WeatherComponent }
];
export const routing = RouterModule.forRoot(routes);
```

Хотя конфигурировать маршруты можно либо в модуле вашего приложения, либо в отдельном файле, последний способ предпочтительнее. Он позволяет использовать конфигурацию маршрутов многократно, чтобы запускать не только приложения, но и сценарии тестов. Сценарий в файле `app.module.ts` приложения погоды задействует в объявлении `@NgModule` константу `routes` (листинг 9.4).

Листинг 9.4. Содержимое файла app.module.ts

```
@NgModule({
  imports: [BrowserModule, HttpModule, ReactiveFormsModule, routing],
  declarations: [AppComponent, HomeComponent, WeatherComponent],
  bootstrap: [AppComponent],
  providers: [
    { provide: LocationStrategy, useClass: HashLocationStrategy },
    { provide: WEATHER_URL_BASE, useValue: 'http://api.openweathermap.org/
    ➥ data/2.5/weather?q=' },
    { provide: WEATHER_URL_SUFFIX, useValue:
    ➥ '&units=imperial&appid=ca3f6d6ca3973a518834983d0b318f73' },
    WeatherService
  ]
})
```

Сценарий теста для маршрутов находится в файле `app.spec.ts`, и в нем повторно используется та же самая константа `routes` (листинг 9.5).

Листинг 9.5. Содержимое файла app.spec.ts

Перед запуском каждого теста выполняется конфигурирование тестового модуля на включение компонентов и поставщиков, необходимых для тестирования маршрутизатора

```
import { TestBed, fakeAsync, inject, tick } from '@angular/core/testing';
import { Location } from '@angular/common';
import { ReactiveFormsModule } from '@angular/forms';
import { provideRoutes, Router } from '@angular/router';
import { RouterTestingModule } from '@angular/router/testing';

import { routes } from '../app.routing';
import { WeatherService } from '../services/weather.service';
import { AppComponent } from './app';
import { HomeComponent } from '../components/home';
import { WeatherComponent } from '../components/weather';

describe('Router', () => {

  beforeEach(() => {
    TestBed.configureTestingModule({

      imports: [ ReactiveFormsModule, RouterTestingModule,
                 RouterTestingModule.withRoutes(routes)],
```

Тестовый набор для маршрутов, определенных в файле app.routing.ts

Предоставление маршрутов для модуля тестирования маршрута

```
    declarations: [AppComponent, HomeComponent, WeatherComponent],
    providers: [{provide: WeatherService, useValue: {} } ]
  ]
 })
});

 it('should be able to navigate to
 ➥ home using commands API',

   fakeAsync(inject([Router, Location], (router: Router, location:
   ➥ Location) => {
     TestBed.createComponent(AppComponent);
     router.navigate(['/']);

     tick();
     expect(location.path()).toBe('/');
   })
 ));

 it('should be able to navigate to weather using commands API',
   fakeAsync(inject([Router, Location], (router: Router, location:
   ➥ Location) => {
     TestBed.createComponent(AppComponent);
     router.navigate(['/weather']);
     tick();
     expect(location.path()).toBe('/weather');
   })
 ));

 it('should be able to navigate to weather by URL',
   fakeAsync(inject([Router, Location], (router: Router, location:
   ➥ Location) =>
     TestBed.createComponent(AppComponent);
     router.navigateByUrl('/weather');
     tick();
     expect(location.path()).toEqual('/weather');
   })
 ));

});
```

Тестирование перехода на WeatherComponent, получающего внедрение WeatherService, следовательно, тут нужно зарегистрировать поставщика для поддельного сервиса

Тестирование возможности маршрутизатора переходить на маршрут /. Он ничего не добавляет к базовому URL, поэтому ожидается, что здесь будет пустая строка

Требуется создание AppComponent, так как в нем объявляется <router-outlet>

Для асинхронного создания AppComponent требуется ускорение времени

Тестирование возможности маршрутизатора переходить на маршрут /weather с использованием метода navigate()

Тестирование возможности маршрутизатора переходить на маршрут /weather с применением метода navigateByUrl()

Обратите внимание на импортирование модуля `ReactiveFormsModule`, поскольку компонент `WeatherComponent` использует API `Forms`.

ПРИМЕЧАНИЕ

Не проводите модульное тестирование в своем приложении кода сторонних поставщиков. В листинге 9.5 в качестве поставщика WeatherService используется пустой объект, который в реальном приложении обращается к удаленному сервису погоды. А что получится, если удаленный сервис в момент запуска тестовой спецификации даст сбой? Модульные тесты позволяют убедиться в работоспособности ваших сценариев, а не в работоспособности сторонних программных средств. Именно поэтому применяется не настоящий сервис WebService, а пустой объект.

При тестировании навигации вашего приложения на клиентской стороне будет использован класс `Router` и его методы `navigate()` и `navigateByUrl()`.

В листинге 9.5 для тестирования правильности обновления адресной строки приложения программными средствами навигации показываются оба метода, и `navigate()`, и `navigateByUrl()`. Но, поскольку в ходе тестирования приложение не запускается, адресной строки браузера не существует, следовательно, она должна быть имитирована. Именно поэтому вместо модуля `RouterModule` используется модуль `RouterTestingModule`, который знает, как проверить ожидаемое содержимое адресной строки, задействуя класс `Location`.

Теперь посмотрим на тестирование внедрения сервисов. Собственно говоря, вы уже внедрили сервисы при тестировании маршрутов:

```
fakeAsync(inject([Router, Location],...))
```

Но в следующем подразделе мы покажем другой способ инициализации требуемых сервисов: будет получен объект типа `Injector` и вызван его метод `get()`.

9.3.3. Тестирование сервиса погоды

Обмен данными с сервисом погоды инкапсулируется в классе `WeatherService` (листинг 9.6).

Листинг 9.6. Содержимое файла weather.service.ts

```
import {Inject, Injectable, OpaqueToken} from '@angular/core';
import {Http, Response} from '@angular/http';
import {Observable} from 'rxjs/Observable';
import 'rxjs/add/operator/filter';
import 'rxjs/add/operator/map';
export const WEATHER_URL_BASE = new OpaqueToken('WeatherUrlBase');
export const WEATHER_URL_SUFFIX = new OpaqueToken('WeatherUrlSuffix');
export interface WeatherResult {
  place: string;
  temperature: number;
  humidity: number;
}
@Injectable()
export class WeatherService {
  constructor(
      private http: Http,
      @Inject(WEATHER_URL_BASE) private urlBase: string,
      @Inject(WEATHER_URL_SUFFIX) private urlSuffix: string) {
  }
  getWeather(city: string): Observable<WeatherResult> {
    return this.http
        .get(this.urlBase + city + this.urlSuffix)
        .map((response: Response) => response.json())
        .filter(this._hasResult)
        .map(this._parseData);
  }
  private _hasResult(data): boolean {
    return data['cod'] !== '404' && data.main;
```

```
  }
  private _parseData(data): WeatherResult {
    let [first,] = data.list;
    return {
      place: data.name || 'unknown',
        temperature: data.main.temp,
        humidity: data.main.humidity
    };
  }
}
```

Обратите внимание на использование типа `OpaqueToken`, упомянутого в главе 4. Он задействован дважды для внедрения в `urlBase` и `urlSuffix` значения, предоставляемого в декораторе `@NgModule`. Применение внедрения зависимостей для `urlBase` и `urlSuffix` упрощает замену при необходимости реального сервиса погоды имитатором.

Показанный в листинге 9.6 метод `getWeather()` формирует URL для HTTP-запроса `get()` путем объединения `urlBase`, `city` и `urlSuffix`. Результат обрабатывается методами `map()`, `filter()` и еще раз методом `map()`, поэтому наблюдаемый объект будет выдавать объекты типа `WeatherResult`.

ПРИМЕЧАНИЕ

Методы _hasResult() и _parseData() не проверяются, поскольку закрытые методы не могут подвергаться модульному тестированию. Если принять решение по их тестированию, то следует изменить их уровень доступности на открытый.

Для тестирования сервиса `WeatherService` будет использоваться класс `MockBackend`, являющийся одной из Angular-реализаций `Http`-объекта. Этот класс не выполняет никаких HTTP-запросов, но перехватывает их и позволяет создавать и возвращать жестко заданные данные в формате ожидаемого результата.

Перед каждым тестом будет получена ссылка на объект типа `Injector`, который станет предоставлять новые экземпляры `MockBackend` и `WeatherService`. Код тестирования последнего находится в файле `weather.service.spec.ts` (листинг 9.7).

Листинг 9.7. Содержимое файла weather.service.spec.ts

```
import {async, getTestBed, TestBed, Injector} from '@angular/core/testing';
import {Response, ResponseOptions, HttpModule, XHRBackend} from '@angular/
➥ http';
import {MockBackend, MockConnection} from '@angular/http/testing';
import {WeatherService, WEATHER_URL_BASE, WEATHER_URL_SUFFIX} from './
➥ weather.service';

describe('WeatherService', () => {
  let mockBackend: MockBackend;

  let service: WeatherService;

  let injector: Injector;
```

```
  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [HttpModule],
      providers: [
        { provide: XHRBackend, useClass: MockBackend },
        { provide: WEATHER_URL_BASE, useValue: '' },
        { provide: WEATHER_URL_SUFFIX, useValue: '' },
        WeatherService
      ]
    });

    injector = getTestBed();
  });

  beforeEach(() => {
    mockBackend = injector.get(XHRBackend);

    service = injector.get(WeatherService);
  });

  it('getWeather() should return weather for New York', async(() => {
    let mockResponseData = {
      cod: '200',
      name: 'New York',
      main: {
        temp: 57,
        humidity: 44
      }
    };

    mockBackend.connections.subscribe((connection: MockConnection) => {
      let responseOpts = new ResponseOptions({body:
      ➥ JSON.stringify(mockResponseData)});
      connection.mockRespond(new Response(responseOpts));
    });

    service.getWeather('New York').subscribe(weather => {
      expect(weather.place).toBe('New York');
      expect(weather.humidity).toBe(44);
      expect(weather.temperature).toBe(57);
    });
  }));
});
```

Получение экземпляра Injector. Интерфейс Injector реализуется в классе TestBed, аметод getTestBed() возвращает объект, реализующий API средства внедрения

Установка поставщиков, которые тестовое средство внедрения должно использовать для токена XHRBackend

Установка поставщиков, которые тестовое средство внедрения должно применять для токена WeatherService

Тест начинается с создания объекта-имитатора для представления погоды в Нью-Йорке. Структура этого объекта подражает фактическим данным ответа от реальной службы погоды

Конфигурирование MockBackend путем подписки на «HTTP-запросы» и имитация реального ответа с содержимым mockResponseData. Тело данного ответа создается с помощью создания экземпляра ResponseOptions

Для возвращения поддельных данных для Нью-Йорка ожидается вызов getWeather('NewYork'). Внутри метода getWeather() используется Http, имитированный MockBackend

Из тестирования внедрения сервисов можно выделить следующие ключевые моменты:

- подготовку поставщиков;
- создание имитаторов при использовании сервисов, выполняющих запросы в адрес внешних серверов.

Способы тестирования навигации и сервисов мы показали, теперь посмотрим, как можно протестировать компонент Angular.

9.3.4. Тестирование компонента погоды

Сервис `WeatherService` внедряется в компонент `WeatherComponent` (`weather.ts`) (листинг 9.8) с помощью конструктора, где происходит подписка на наблюдаемые сообщения, поступающие от `WeatherService`. Когда пользователь начинает вводить название города в пользовательском интерфейсе, вызывается метод `getWeather()` и возвращенные данные о погоде отображаются в шаблоне через привязку.

Листинг 9.8. Содержимое файла weather.ts

```
import {Component} from '@angular/core';
import {FormControl} from '@angular/forms';
import 'rxjs/add/operator/debounceTime';
import 'rxjs/add/operator/switchMap';
import {WeatherService, WeatherResult} from '../services/weather.service';
@Component({
  selector: 'my-weather',
  template: `
    <h2>Weather</h2>
    <input type="text" placeholder="Enter city" [formControl]="searchInput">
    <h3>Current weather in {{weather?.place}}:</h3>
    <ul>
      <li>Temperature: {{weather?.temperature}}F</li>
      <li>Humidity: {{weather?.humidity}}%</li>
    </ul>
  `,
})
export class WeatherComponent {
  searchInput: FormControl;
  weather: WeatherResult;
  constructor(weatherService: WeatherService) {
    this.searchInput = new FormControl('');
    this.searchInput.valueChanges
        .debounceTime(300)
        .switchMap((place: string) => weatherService.getWeather(place))
        .subscribe(
            (wthr: WeatherResult) => this.weather = wthr,
            error => console.error(error),
            () => console.log('Weather is retrieved'));
  }
}
```

Желательно написать тест для проверки того, что при получении значений свойством `weather` шаблон соответственно обновляется с помощью привязки. Желательно также проверить, что при изменении значения объекта `searchInput` наблюдаемый объект выдает данные через свое свойство `valueChanges`.

Elvis-оператор

Шаблон компонента `WeatherComponent` включает выражения со знаками вопроса, например, `weather?.place`. В этом контексте знак вопроса называется Elvis-оператором.

Свойство `weather` заполняется в асинхронном режиме, и если на момент вычисления выражения это `null`, то выражение `weather.place` выдаст ошибку. Для подавления `null`-разыменования используется Elvis-оператор, чтобы создать короткое замыкание на дополнительное вычисление, если значением `weather` является `null`. Elvis-оператор предлагает явную запись, показывающую, какие значения могут быть вычислены в `null`.

Тестовый набор будет включать один тест для проверки ожидаемой работоспособности привязки данных. Код `TestBed.createComponent(WeatherComponent);` создаст приспособление `ComponentFixture`, содержащее ссылки на `WeatherComponent`, а также на DOM-объект, представляющий этот компонент. В листинге 9.8 свойство `weather` используется для привязок; оно инициализируется литералом объекта, который содержит жестко заданные значения для места, влажности и температуры (`place`, `humidity` и `temperature`) .

После этого будет принудительно обнаружено изменение путем вызова метода `detectChanges()`, принадлежащего экземпляру класса `ComponentFixture`. Появление значений от `weather` ожидается в шаблоне компонента в одном теге `<h3>` и двух тегах `<li>`. Код для данного теста находится в файле `weather.spec.ts` (листинг 9.9).

Листинг 9.9. Содержимое файла weather.spec.ts

```
import { TestBed} from '@angular/core/testing';
import { ReactiveFormsModule } from '@angular/forms';

import { WeatherComponent } from './weather';
import { WeatherService } from '../services/weather.service';

describe('WeatherComponent', () => {

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [ ReactiveFormsModule ],
      declarations: [ WeatherComponent],
      providers: [{provide: WeatherService, useValue: {} }]
    })
  });

  it('should display the weather ', () => {
    let fixture = TestBed.createComponent(WeatherComponent);

    let element = fixture.nativeElement;

    let component = fixture.componentInstance;

    component.weather = {place: 'New York',
                         humidity: 44, temperature: 57};
```

Ожидается, что в компонент WeatherComponent будет внедрен сервис WeatherService, и реальный сервис будет заменен пустым объектом

Получаем ссылку на тестируемый компонент

Создается экземпляр WeatherComponent, а обратно получаем ComponentFixture

Получаем ссылку на HTML-элемент, выведенный для этого компонента

Инициализируем свойство компонента weather, как будто данные поступили с сервера

```
    fixture.detectChanges();

    expect(element.querySelector('h3').innerHTML).toBe('Current weather in
    ➥ New York:');

    expect(element.querySelector('li:nth-of
     type(1)').innerHTML).toBe('Temperature: 57F');
    expect(element.querySelector('li:nth-of
     type(2)').innerHTML).toBe('Humidity: 44%');
  });
});
```

Инициируем обнаружение изменений

Сравниваем текст первого и второго элементов <li> с ожидаемыми значениями. Для получения текста элемента <li> по его позиции используем CSS-селектор li:nth-of-type()

Сравниваем текст элемента <h3> с ожидаемым значением

СОВЕТ

В листинге 9.9 для имитации WeatherService используется пустой объект, поскольку в отношении него вызов каких-либо методов не планируется. Определить сервис-имитатор можно в виде класса MockWeatherService, реализующего WeatherService и предоставляющего реализацию реальных методов, но они будут возвращать жестко заданные значения. При определении сервиса-имитатора в приложениях, предназначенных для реальной работы, целесообразно создавать классы, реализующие интерфейсы настоящих сервисов.

ПРИМЕЧАНИЕ

В главе 7 были рассмотрены два подхода к созданию форм в Angular. Хотя шаблон-ориентированный подход требует меньшего объема кода, применение реактивных форм делает их более пригодными для тестирования, не требуя для этого DOM-объект.

Запуск тестов в средстве запуска на основе HTML. Запустим тестовый набор для погодного приложения в средстве запуска на основе HTML. Для этого нужно просто запустить живой сервер и ввести в адресную строку браузера URL http://localhost:8080/test.html. Все тесты должны быть пройдены, и окно браузера должно выглядеть так же, как на рис. 9.4.

При написании тестов хочется увидеть, как они могут быть не пройдены. Устроим так, чтобы один тест не был пройден, для наблюдения того, как будет сообщено об этом. Измените температуру в строке, где инициализируется свойство `weather`, на 58 градусов:

```
component.weather = {place: 'New York', humidity: 44, temperature: 58};
```

А тест по-прежнему будет ожидать, что пользовательский интерфейс выведет на экран температуру 57 градусов:

```
expect(element.querySelector('li:nth-of-type(1)').innerHTML)
➥ .toBe('Temperature: 57F');
```

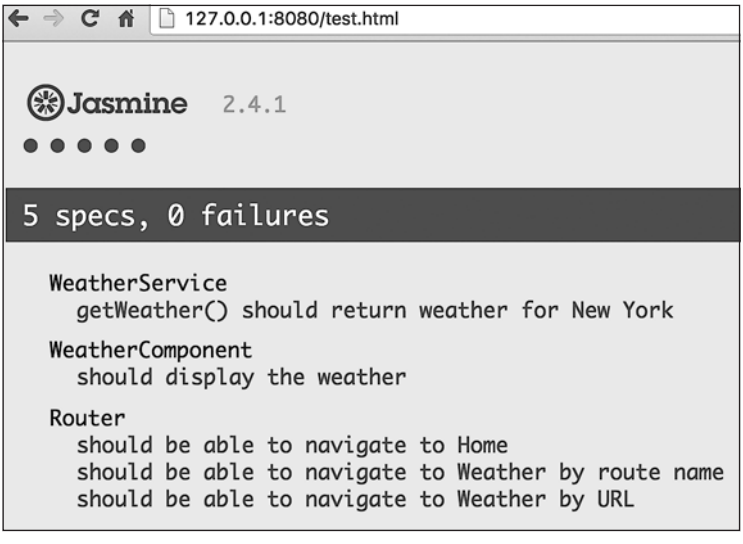


Рис. 9.4. Тесты пройдены

Вывод результатов тестирования, показанный на рис. 9.5, сообщает о сбое одного из пяти тестов.

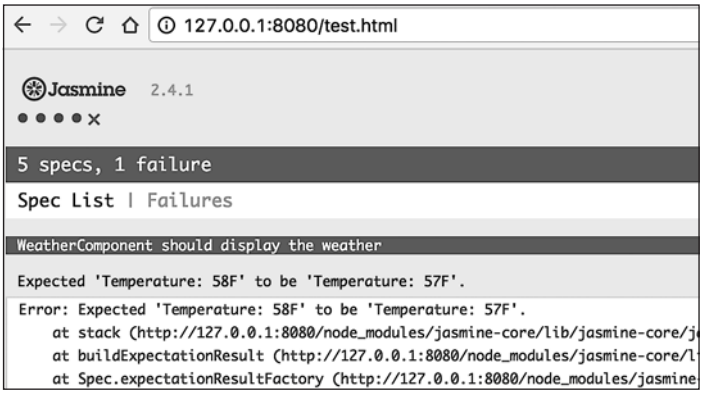


Рис. 9.5. Тесты не пройдены

Запуск тестов в браузере вручную — не самый лучший способ модульного тестирования кода. Нужен процесс тестирования, запускаемый как сценарий из командной строки, чтобы его можно было встроить в автоматизированный процесс сборки. В Jasmine имеется средство запуска, которое может использоваться из приглашения командной строки. Но предпочтительнее задействовать независимое средство для запуска тестов под названием Karma, способное работать с множеством различных сред модульного тестирования. Порядок применения этой программы будет рассмотрен в следующем разделе.

9.4. Запуск тестов с помощью Karma

Karma (https://karma-runner.github.io/1.0/index.html) — средство для запуска тестов, изначально созданное командой разработчиков, но использовавшееся для тестирования кода JavaScript, написанного с применением или без применения какой-либо среды разработки. Это средство создано с помощью Node.js, и, хотя оно не запускается в браузере, у него имеется возможность запускать тесты, чтобы проверить, будет ли ваше приложение работать в нескольких браузерах (вы будете запускать тесты для Chrome и Firefox).

Применительно к приложению Weather вы установите Karma и дополнительные модули для Jasmine, Chrome и Firefox и сохраните их в разделе `devDependencies` файла `package.json`:

```
npm install karma karma-jasmine karma-chrome-launcher karma-firefox-
➥ launcher --save-dev
```

Для запуска Karma настройте в npm команду `test` вашего проекта следующим образом:

```
"scripts": {
  "test": "karma start karma.conf.js"
}
```

ПРИМЕЧАНИЕ

Исполнительный файл karma имеет двоичную природу и находится в каталоге node_modules/.bin.

Кроме того, вы создадите небольшой конфигурационный файл `karma.conf.js` (листинг 9.10), позволяющий Karma узнать о проекте. Этот файл будет находиться в корневом каталоге проекта и включать пути к файлам Angular, а также настройки конфигурации для средства запуска тестов Karma.

Листинг 9.10. Содержимое файла karma.conf.js

```
Однократный запуск тестов и остановка. Эта настройка пригодится при запуске
Karma в качестве составной части автоматизированной сборки

module.exports = function (config) {
  config.set({
    browsers: ['Chrome', 'Firefox'],

    frameworks: ['jasmine'],

    reporters: ['dots'],

    singleRun: true,

    files: [
      // Пути, загруженные Karma
      'node_modules/typescript/lib/typescript.js',
      'node_modules/reflect-metadata/Reflect.js',
      'node_modules/systemjs/dist/system.src.js',
      'node_modules/zone.js/dist/zone.js',
      'node_modules/zone.js/dist/async-test.js',
```

Тестирование работы приложения в браузерах Chrome и Firefox

Блочные тесты, написанные с использованием Jasmine

Вывод точек в консоли для обозначения процесса тестирования

Средство Karma должно знать о файлах из среды Angular (включая библиотеку тестирования)

```
      'node_modules/zone.js/dist/fake-async-test.js',
      'node_modules/zone.js/dist/long-stack-trace-zone.js',
      'node_modules/zone.js/dist/proxy.js',
      'node_modules/zone.js/dist/sync-test.js',
      'node_modules/zone.js/dist/jasmine-patch.js',

      // Пути, загруженные импортированными модулями
      {pattern: 'karma-systemjs.config.js',
       ➥ included: true, watched: false},
      {pattern: 'karma-test-runner.js', included: true, watched: false},
      {pattern: 'node_modules/@angular/**/

    *.js', included: false, watched: false},
     {pattern: 'node_modules/@angular/**/
    *.js.map', included: false, watched: false},
     {pattern: 'node_modules/rxjs/**/
    *.js', included: false, watched: false},
     {pattern: 'node_modules/rxjs/**/

    *.js.map', included: false, watched: false},
     {pattern: 'app/**/*.ts',
      ➥ included: false, watched: true}
  ],

  proxies: {
    '/app/': '/base/app/'
  },
  plugins: [
    'karma-jasmine',
    'karma-chrome-launcher',
    'karma-firefox-launcher'
  ]
 })
};
```

Конфигурация KarmaSystemJS. Идентична файлу systemjs.config.js, но дополнительно определяет baseURL: 'base'

Этот сценарий запустит тесты

Файлы загружаются путем импортирования модуля и могут содержать либо сценарии тестирования, либо код приложения, включенный в операторы import

Требуется для функциональных средств компонента, вызываемых компилятором Angular. Имена файлов, начинающиеся с /app в свойствах styleUrls и templateUrl (не используемые в погодном приложении), должны быть представлены через генерируемый Karma путь /base/app

Дополнительные модули, требуемые для запуска

В большей части листинга 9.10 перечисляются пути нахождения требуемых файлов. Karma создает временную HTML-страницу, включающую файлы, перечисленные с `included: true`. Файлы, перечисленные с `included: false`, будут динамически подгружаться в ходе выполнения. Все Angular-файлы, включая тестируемые, загружаются в динамическом режиме с использованием SystemJS.

Вам следует добавить к проекту еще один файл: `karma-test-runner.js` (листинг 9.11). Это сценарий, который фактически запускает тесты.

Листинг 9.11. Содержимое файла karma-test-runner.js

```
Error.stackTraceLimit = Infinity;

jasmine.DEFAULT_TIMEOUT_INTERVAL = 1000;

__karma__.loaded = function () {};
```

Позволяет браузеру показывать полную трассировку стека при ошибке

Исходное истечение срока ожидания Jasmine для вызова асинхронной функции составляет пять секунд, но в этой строке оно изменяется на одну секунду

Поскольку код приложения и спецификаций загружается в асинхронном режиме, средство Karma не должно запускаться при выдаче события loaded. Вызов karma.start() будет осуществляться позже, как только загрузятся все спецификации

```
function resolveTestFiles() {

  return Object.keys(window.__karma__.files)
      .filter(function (path) { return /\.spec\.ts$/.test(path); })
      .map(function (moduleName) { return System.import(moduleName); });
}

Promise.all([
    System.import('@angular/core/testing'),
    System.import('@angular/platform-browser-dynamic/testing')
  ]).
  then(function (modules) {
    var testing = modules[0];
    var browser = modules[1];

    testing.TestBed.initTestEnvironment(
        browser.BrowserDynamicTestingModule,
        browser.platformBrowserDynamicTesting());
  }).
  then(function () { return Promise.all(resolveTestFiles()); })
 .then(function () { __karma__.start(); },
       function (error) { __karma__.error(error.stack || error); });
```

Поиск всех файлов с расширениями имен spec.ts

Загрузка двух Angular-модулей, требуемых для тестирования

Указание после загрузки модулей исходных поставщиков Angular

Инициализация среды тестирования

Загрузка тестовых спецификаций

Запуск тестов

Теперь все готово к запуску ваших тестов с использованием в командной строке команды `npm test`. В ходе своей работы Karma будет открывать и закрывать каждый указанный в конфигурации браузер и выводить на экран результаты тестирования, как показано на рис. 9.6.

```
test_samples — -bash — 80×24
Yakov-2:test_samples yfain11$ npm test

> test_samples@ test /Users/yfain11/Documents/core-angular2/code/chapter9/test_s
amples
> karma start karma.conf.js

23 03 2016 08:28:22.421:INFO [karma]: Karma v0.13.19 server started at http://lo
calhost:9876/
23 03 2016 08:28:22.434:INFO [launcher]: Starting browser Chrome
23 03 2016 08:28:22.440:INFO [launcher]: Starting browser Firefox
23 03 2016 08:28:23.813:INFO [Chrome 49.0.2623 (Mac OS X 10.11.3)]: Connected on
 socket /#PaN8abvrx9zT6TdoAAAA with id 62891017
.....
Chrome 49.0.2623 (Mac OS X 10.11.3): Executed 5 of 5 SUCCESS (0.17 secs / 0.138
secs)
23 03 2016 08:28:25.820:INFO [Firefox 45.0.0 (Mac OS X 10.11.0)]: Connected on s
ocket /#vIS5ZKxne4OSArBZAAAB with id 23889967
.....
Firefox 45.0.0 (Mac OS X 10.11.0): Executed 5 of 5 SUCCESS (0.099 secs / 0.086 s
ecs)
TOTAL: 10 SUCCESS
```

Рис. 9.6. Тестирование погодного приложения с использованием Karma

Разработчики склонны применять самые последние версии браузеров, имеющих наиболее совершенные инструменты; именно таким является Google Chrome. Нам попадались реальные проекты, в ходе реализации которых разработчиком демонстрировалась великолепная работа приложения в браузере Chrome, а затем пользователи жаловались на ошибки проекта, проявлявшиеся в Safari. Нужно обеспечить использование в процессе разработки Karma и тестировать приложение во всех браузерах. Перед передачей приложения команде по проверке качества или перед его демонстрацией своему руководителю следует убедиться, что средство для запуска тестов Karma не сообщило об ошибках во всех востребованных браузерах.

Мы закончили обзор создания и запуска модульных тестов, теперь приступим к реализации тестов для онлайн-аукциона.

9.5. Практикум: модульное тестирование онлайн-аукциона

Цель этого практикума — демонстрация модульного тестирования отдельно взятых модулей приложения онлайн-аукциона. В частности, модульные тесты будут добавлены для `ApplicationComponent`, `StarsComponent` и `ProductService`. Тесты запустятся с использованием имеющегося в Jasmine средства запуска тестов на основе HTML, а затем с применением Karma.

ПРИМЕЧАНИЕ

> В качестве отправной точки мы собираемся воспользоваться приложением-аукционом из главы 8, поэтому его нужно скопировать в отдельный каталог и следовать инструкциям, рассматриваемым в данном разделе. Если вы больше склоняетесь к просмотру кода, а не к его набору, то задействуйте код, предоставляемый с главой 9, и запустите тесты.

Установите Jasmine, Karma, файлы определения типов для Jasmine и все зависимости Angular, запустив следующие команды:

```
npm install jasmine-core karma karma-jasmine karma-chrome-launcher karma-
➥ firefox-launcher --save-dev
npm install @types/jasmine --save-dev
npm install
```

В клиентском каталоге создайте новый файл `auction-unit-tests.html` для загрузки тестов Jasmine (листинг 9.12).

Листинг 9.12. Содержимое файла auction-unit-tests.html

```
<!DOCTYPE html>
<html>
<head>
  <title>[TEST] Online Auction</title>
  <!-- браузерный компилятор TypeScript -->
  <script src="node_modules/typescript/lib/typescript.js"></script>
```

```
  <!-- Полифиллы -->
  <script src="node_modules/reflect-metadata/Reflect.js"></script>
  <!-- Jasmine -->
  <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>
  <!-- Zone.js -->
  <script src="node_modules/zone.js/dist/zone.js"></script>
  <script src="node_modules/zone.js/dist/proxy.js"></script>
  <script src="node_modules/zone.js/dist/sync-test.js"></script>
  <script src="node_modules/zone.js/dist/jasmine-patch.js"></script>
  <script src="node_modules/zone.js/dist/async-test.js"></script>
  <script src="node_modules/zone.js/dist/fake-async-test.js"></script>
  <script src="node_modules/zone.js/dist/long-stack-trace-zone.js"></script>
  <!-- SystemJS -->
  <script src="node_modules/systemjs/dist/system.src.js"></script>
  <script src="systemjs.config.js"></script>
</head>
<body>
<script>
  var SPEC_MODULES = [
    'app/components/application/application.spec',
    'app/components/stars/stars.spec',
    'app/services/product-service.spec'
  ];
  Promise.all([
    System.import('@angular/core/testing'),
    System.import('@angular/platform-browser-dynamic/testing')
  ])
          .then(function (modules) {
            var testing = modules[0];
            var browser = modules[1];
            testing.TestBed.initTestEnvironment(
                    browser.BrowserDynamicTestingModule,
                    browser.platformBrowserDynamicTesting());
            // загрузка всех файлов спецификаций
            return Promise.all(SPEC_MODULES.map(function (module) {
              return System.import(module);
            }));
          })
          .then(window.onload)
          .catch(console.error.bind(console));
</script>
</body>
</html>
```

Содержимое этого файла похоже на содержимое файла `test.html` из погодного приложения. Единственное отличие заключается в том, что вы загружаете здесь другие файлы спецификаций: `application.spec`, `stars.spec` и `product-service.spec`.

9.5.1. Тестирование ApplicationComponent

Чтобы протестировать успешность создания экземпляра компонента `ApplicationComponent`, создайте в каталоге `client/app/components/application` файл `application.spec.ts` (листинг 9.13). Это, конечно, не самый полезный тест, но он может послужить иллюстрацией того, успешно ли создан экземпляр класса TypeScript (даже не связанный с Angular).

Листинг 9.13. Содержимое файла application.spec.ts

```
import ApplicationComponent from './application';
describe('ApplicationComponent', () => {
  it('is successfully instantiated', () => {
    const app = new ApplicationComponent();
    expect(app instanceof ApplicationComponent).toEqual(true);
  });
});
```

9.5.2. Тестирование ProductService

Чтобы протестировать `ProductService`, создайте в каталоге `app/services` файл `product-service.spec.ts` (листинг 9.14). В этой спецификации будет тестироваться HTTP-сервис и, хотя функция `it()` совсем небольшая, перед запуском теста предстоит множество подготовительных действий.

Листинг 9.14. Содержимое файла product-service.spec.ts

```
import {async, getTestBed, TestBed, inject, Injector} from '@angular/core/
➥ testing';
import {Response, ResponseOptions, HttpModule, XHRBackend} from '@angular/
➥ http';
import {MockBackend, MockConnection} from '@angular/http/testing';
import {ProductService} from './product-service';

describe('ProductService', () => {
  let mockBackend: MockBackend;
  let service: ProductService;

  let injector: Injector;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [HttpModule],
      providers: [
        { provide: XHRBackend, useClass: MockBackend },
        ProductService
      ]
    });
    injector = getTestBed();
  });
```

MockBackend служит в качестве реализации HTTP-сервиса. А MockConnection представляет подключение

Сохранение ссылок на внедренные сервисы, чтобы тест мог ими воспользоваться

Переопределение исходной реализации Http-объекта путем явного создания экземпляра этого объекта и передачи ему в качестве аргумента MockBackend. Изменения в BaseRequestOption не вносятся, но это обязательный аргумент

```
  beforeEach(inject([XHRBackend, ProductService], (_mockBackend,
    ➥ _service) => {
    mockBackend = _mockBackend;
    service = _service;
  }));

  it('getProductById() should return Product with ID=1', async(() => {

    let mockProduct = {id: 1};

    mockBackend.connections.subscribe((connection: MockConnection) => {
      let responseOpts = new ResponseOptions({body:
      ➥ JSON.stringify(mockProduct)});
      connection.mockRespond(new Response(responseOpts));
  });

    service.getProductById(1).subscribe(p => {
       expect(p.id).toBe(1);
    });
  }));
});
```

Подготовка фиктивных данных, возвращаемых MockBackend

Конфигурирование имитатора серверной части

Вызов getProductById(1) в отношении сервиса должен возвратить объект с идентификатором, равным 1

Сначала создается литерал объекта, `mockProduct = {id: 1}`, используемый для имитации данных, которые могут поступить с сервера в качестве HTTP-ответа. Нужно, чтобы `mockBackend` выполнил имитацию и осуществил возврат объекта с жестко заданными значениями для каждого HTTP-запроса. Вы могли бы создать экземпляр `Product` и с более богатыми свойствами, но для этого простого теста достаточно наличия одного идентификатора.

9.5.3. Тестирование StarsComponent

Для последнего теста мы выбрали `StarsComponent`, поскольку в нем демонстрируется, как можно протестировать свойства компонента и средство выдачи события. Компонент `StarsComponent` загружает свой HTML из файла, для которого в процессе тестирования требуется специальная обработка. Angular загружает файлы, указанные в `templateUrl` в асинхронном режиме, и выполняет их компиляцию к нужному моменту. Вам потребуется сделать то же самое в спецификации теста путем вызова метода `TestBed.compileComponents()`. Этот шаг необходим для любого компонента, использующего свойство `templateUrl`. Создайте в каталоге `client/app/components/stars` файл `stars.spec.ts` со следующим содержимым (листинг 9.15).

Листинг 9.15. Файл stars.spec.ts

```
import { TestBed, async, fakeAsync, inject } from '@angular/core/testing';
import StarsComponent from './stars';

describe('StarsComponent', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({
```

```
      declarations: [ StarsComponent ]
    });
  });

  beforeEach(async(() => {
    TestBed.compileComponents();
  }));

  it('is successfully injected', () => {
    let component =
        TestBed.createComponent(StarsComponent).componentInstance;
    expect(component instanceof StarsComponent).toEqual(true);
  });

  it('readonly property is true
  ➥ by default', () => {
    let component =
        TestBed.createComponent(StarsComponent).componentInstance;
    expect(component.readonly).toEqual(true);
  });

  it('all stars are empty', () => {
    let fixture = TestBed.createComponent(StarsComponent);
    let element = fixture.nativeElement;
    let cmp = fixture.componentInstance;
    cmp.rating = 0;
    fixture.detectChanges();

    let selector = '.glyphicon-star-empty';
    expect(element.querySelectorAll(selector).length).toBe(5);
  });

  it('all stars are filled', () => {
    let fixture = TestBed.createComponent(StarsComponent);
    let element = fixture.nativeElement;
    let cmp = fixture.componentInstance;
    cmp.rating = 5;

    fixture.detectChanges();

    let selector = '.glyphicon-star:not(.glyphicon-star-empty)';
    expect(element.querySelectorAll(selector).length).toBe(5);
  });

  it('emits rating change event when readonly is false', async(() => {
    let component =
        TestBed.createComponent(StarsComponent).componentInstance;
    component.ratingChange.subscribe(r => {
      expect(r).toBe(3);
    });
    component.readonly = false;
    component.fillStarsWithColor(2);
  }));
});
```

Компиляция содержимого файла, используемого в templateUrl

Проверка факта внедрения экземпляра (сравните с листингом 9.13)

Создание приспособления и получение ссылки на экземпляр компонента

Проверка факта наличия исходного значения true у входного свойства readonly компонента StarsComponent. Пользователь может выбирать звезды лишь в режиме LeaveReview (Оставить отзыв)

Проверка того, что при нулевом рейтинге все звезды выводятся пустыми

Проверка того, что при рейтинге, равном пяти, все звезды выводятся заполненными

Проверка работоспособности объекта EventEmitter

Класс TestBed создает новый экземпляр компонента StarsComponent (здесь внедренный экземпляр не используется) и предоставляет приспособление со ссылками на компонент и на исходный элемент. Для проверки того, что все звезды пустые, входному свойству rating экземпляра компонента присваивается нуль. Фактически свойство rating является в StarsComponent сеттером, модифицирующим как rating, так и массив stars:

```
set rating(value: number) {
  this._rating = value || 0;
  this.stars = Array(this.maxStars).fill(true, 0, this.rating);
}
```

Затем запускается цикл обнаружения изменений, заставляющий цикл *ngFor заново выполнять вывод изображений в виде звезд в шаблоне StarsComponent:

```
<p>
  <span *ngFor="let star of stars; let i = index"
       class="starrating glyphicon glyphicon-star"
       [class.glyphicon-star-empty]="!star"
       (click)="fillStarsWithColor(i)">
  </span>
  <span *ngIf="rating">{{rating | number:'.0-2'}} stars</span>
</p>
```

Кодом CSS для заполненных звезд является starrating glyphicon glyphicon-star. У пустых звезд имеется дополнительный класс CSS, glyphicon-star-empty. Тест на пустоту всех звезд — 'all stars are empty' — использует селектор glyphicon-star-empty и ожидает, что имеется именно пять исходных элементов с таким классом.

Тест на заполненность всех звезд — 'all stars are filled' — присваивает рейтинг 5. В нем применяется CSS-селектор .glyphicon-star:not(.glyphicon-star-empty), в котором оператор not задействован для подтверждения того, что звезды не пусты.

Тест на выдачу события изменения рейтинга, когда readonly имеет значение false, — 'emits rating change event when readonly is false' — использует внедренный компонент. В нем выполняется подписка на событие ratingChange в ожидании, что значением рейтинга будет 3. Когда пользователь хочет изменить рейтинг, он нажимает третью звезду (оставляя отзыв), тем самым в отношении компонента вызывается метод заполнения звезд цветом — fillStarsWithColor, с передачей 3 в качестве аргумента index:

```
fillStarsWithColor(index) {
  if (!this.readonly) {
    this.rating = index + 1; // для предотвращения нулевого рейтинга
    this.ratingChange.emit(this.rating);
  }
}
```

Поскольку никакой пользователь во время модульного тестирования кнопку мыши не нажимает, этот метод вызывается программным путем:

```
component.readonly = false;
 component.fillStarsWithColor(2);
```

Если нужно посмотреть, как этот тест не будет пройден, то измените аргумент `fillStarsWithColor()` на любое число, отличное от двух.

Порядок операций в тестировании событий

В коде теста `'emits rating change event when readonly is false'` то обстоятельство, что предыдущие две строки помещены в конце теста, после вызова `subscribe()`, может вызвать удивление. Выполнение подписки на `Observable` имеет ленивый характер, и следующий элемент будет получен только после вызова метода `fillStarsWithColor(2)`, что приведет к выдаче события. Если переместить вызов метода `subscribe()` вниз, то событие будет выдано еще до создания подписчика и тест не будет пройден по истечении времени ожидания, поскольку метод `done()` никогда не будет вызван.

9.5.4. Запуск тестов

Чтобы запустить тесты, сначала нужно провести повторную компиляцию серверного кода, запустив команду `npm run tsc`. Затем следует запустить приложение-аукцион, введя в консоли команду `npm start`. Тем самым на порте 8000 будет запущен Node-сервер. После ввода в браузер адреса http://localhost:8000/auction-unit-tests.html тесты должны запуститься и произвести вывод, показанный на рис. 9.7.

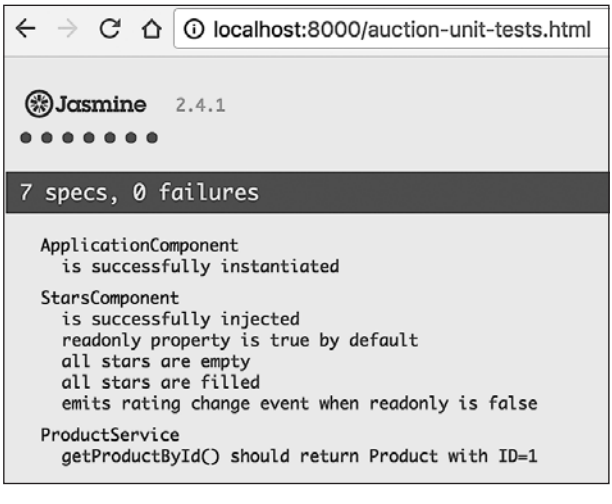


Рис. 9.7. Тестирование онлайн-аукциона с использованием средства запуска тестов на основе HTML

Для запуска этих же тестов с использованием Karma скопируйте файлы `karma.conf` и `karma-test-runner` из каталога `auction` главы 9 в корневой каталог вашего проекта. (Данные файлы рассматривались в разделе 9.4.) Запустите команду `npm test` и увидите вывод, показанный на рис. 9.8.

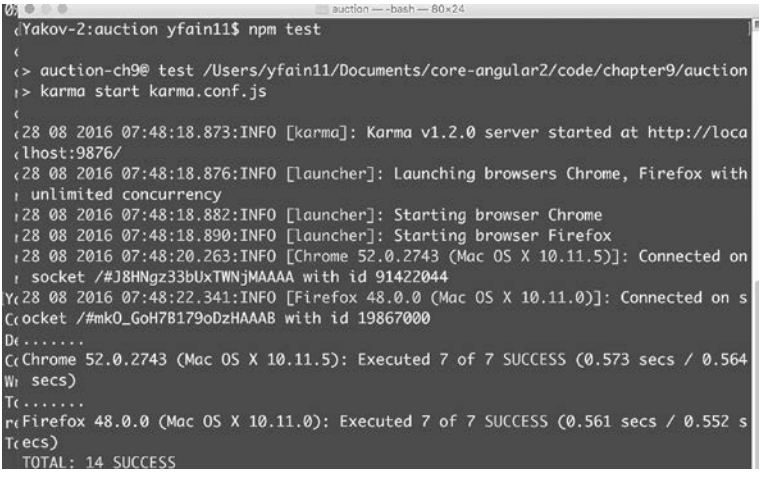


Рис. 9.8. Тестирование аукциона с использованием Karma

9.6. Резюме

Важность модульного тестирования Angular-приложений трудно переоценить. Модульные тесты позволяют убедиться в том, что каждый компонент или сервис приложения работают в точном соответствии с вашими ожиданиями. В данной главе был показан порядок создания модульных тестов с использованием среды Jasmine, а также продемонстрированы приемы их запуска с применением либо Jasmine, либо Karma.

Вот основные выводы этой главы.

- Хорошими кандидатами на создание тестового набора являются компонент или сервис.
- Хотя все файлы тестов можно хранить отдельно от приложения, удобнее всего будет хранить их рядом с тестируемым компонентом.
- Модульные тесты выполняются очень быстро, и ими должна быть протестирована основная часть логики функционирования приложения.
- При создании тестов создавайте условия, при которых код их не проходит, чтобы убедиться, что в сообщениях о сбое будет нетрудно разобраться.
- Если будет принято решение о реализации сквозного тестирования, то в ходе данного тестирования повторно запускать модульные тесты не нужно.
- Запуск модульных тестов должен стать частью вашего процесса автоматизированной сборки. Как это делается, будет показано в главе 10.

10 Упаковка и развертывание приложений с помощью Webpack

В этой главе:

- упаковка приложений для развертывания с использованием Webpack;
- конфигурирование Webpack для упаковки Angular-приложений в dev и prod;
- встраивание средства для запуска тестов Karma в автоматизированный процесс сборки;
- создание сборки prod для онлайн-аукциона;
- автоматизация создания и упаковки с использованием Anguar CLI.

В процессе чтения книги вы создали и развернули множество версий онлайн-аукциона и большое количество менее крупных приложений. Веб-серверы вполне справлялись с обслуживанием ваших приложений при их работе с пользователем. Так почему бы просто не скопировать все файлы приложения на производственный сервер и не запустить команду `npm install`, решив тем самым все вопросы с развертыванием?

Независимо от того, какой язык программирования и какую среду вы используете, нужно приложить усилия для достижения двух целей:

- развертываемое веб-приложение должно быть небольшим по размеру (чтобы его можно было быстрее загрузить);
- при запуске браузер должен обойтись минимумом запросов к серверу (для скорейшей загрузки).

Когда браузер выполняет запрос к серверу, он получает HTML-документы, которые могут включать дополнительные файлы: CSS-таблицы, изображения, видеоклипы и т. д. Возьмем для примера онлайн-аукцион. В процессе запуска он отправляет сотни запросов к серверу, просто чтобы загрузить Angular с его зависимостями и компилятор TypeScript; в целом это составляет по объему 5,5 Мбайт. Добавьте сюда созданный вами код, составляющий пару десятков файлов HTML, TypeScript и CSS, не говоря уже об изображениях! Для такого небольшого при-

ложения получается очень много загружаемого кода и слишком много запросов к серверу. Посмотрите на рис. 10.1, где показано содержимое вкладки Network (Сеть) панели Developer Tools (Инструменты разработчика) браузера Chrome после загрузки аукциона в ваш браузер: там огромное количество сетевых запросов и громадный объем приложения.

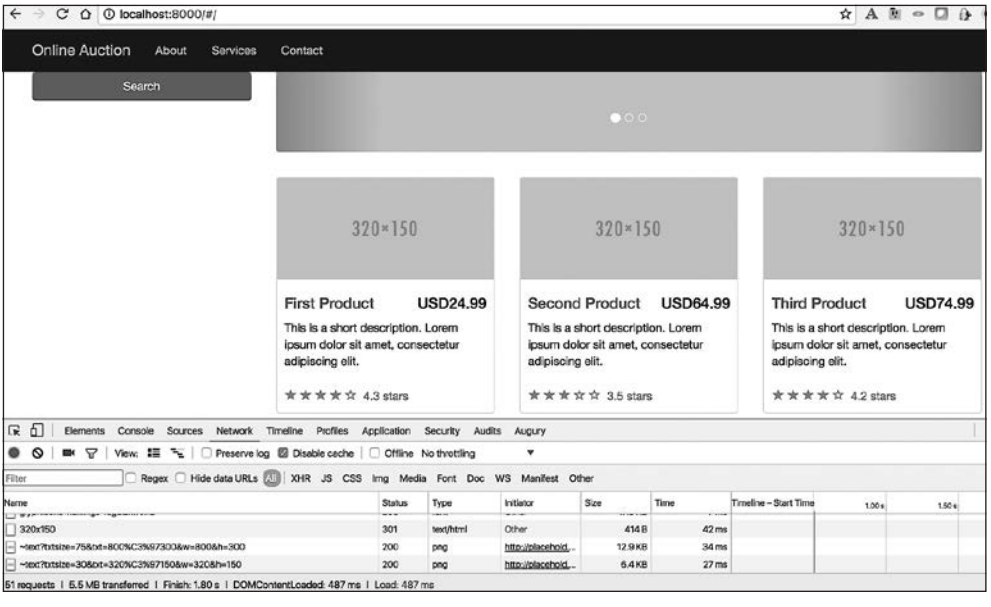


Рис. 10.1. Отслеживание версии развертывания приложения онлайн-аукциона

Настоящие приложения содержат сотни и даже тысячи файлов, требующих минимизации, оптимизации и создания из них единого пакета. Кроме того, для коммерческого применения приложения можно предварительно откомпилировать код в JavaScript, избавившись от необходимости загрузки в браузер компилятора TypeScript объемом 3 Мбайт.

Для развертывания веб-приложений на JavaScript имеется несколько популярных инструментальных средств. Все они используют Node и доступны в виде npm-пакетов. Эти средства можно отнести к двум основным категориям:

- средства запуска задач;
- загрузчики и упаковщики модулей.

К весьма распространенным универсальным средствам запуска задач можно отнести Grunt (http://gruntjs.com) и Gulp (http://gulpjs.com). Они совершенно не разбираются в приложениях JavaScript, но позволяют настраивать и запускать задачи, необходимые для развертывания приложений. Приспособить Grunt и Gulp под процессы сборки не так-то просто, поскольку их конфигурационные файлы имеют длину в несколько сотен строк.

В данной книге для запуска задач использовались npm-сценарии, а под задачей понимался сценарий или двоичный файл, который мог быть выполнен из командной строки. Настроить npm-сценарии намного проще, чем применять Grunt и Gulp, и мы продолжим задействовать эти сценарии в данной главе. Когда ваш проект станет сложнее, а количество npm-сценариев превысит разумные пределы, позволяющие с ними справиться, можно будет рассмотреть вопрос использования для запуска сборок Grunt или Gulp.

До сих пор для загрузки модулей применялось такое средство, как SystemJS. А к популярным упаковщикам можно отнести Browserify (http://browserify.org), Webpack (http://webpack.github.io), Broccoli (www.npmjs.com/package/broccolisystem-builder) и Rollup (http://rollupjs.org). Каждое из этих средств создает пакеты кода, используемые браузером. Самым простым, позволяющим конвертировать и объединять все ресурсы вашего приложения в пакеты с минимальными настройками, является Webpack. Краткое сравнение различных упаковщиков доступно на http://webpack.github.io/docs/comparison.html.

Упаковщик Webpack был создан специально для веб-приложений, запускаемых в браузере, и многие обычные задачи, необходимые для подготовки сборок веб-приложений, поддерживаются изначально с минимальными настройками конфигурации и без необходимости установки дополнительных модулей. Эта глава начинается с введения в Webpack, а затем вы приготовите две разные сборки (dev и prod) для онлайн-аукциона. И наконец, запустите оптимизированную версию онлайн-аукциона и сравните размер приложения с тем, что показано на рис. 10.1.

Средство SystemJS в этой главе использоваться не будет: Webpack станет вызывать компилятор TypeScript в процессе пакетирования приложений. Процесс компиляции будет управляться специальным загрузчиком, внутри которого для транспиляции TypeScript в JavaScript используется tsc.

ПРИМЕЧАНИЕ

Командой разработчиков Angular создан интерфейс командной строки Angular CLI (https://github.com/angular/angular-cli) для автоматизации создания, тестирования и развертывания приложения. Angular CLI использует упаковщик Webpack на внутреннем уровне. Данный интерфейс будет представлен в этой главе чуть позже.

10.1. Знакомство с Webpack

Готовясь в путешествие, можно упаковать в пару чемоданов десятки разных вещей. Смекалистые путешественники используют специальные пакеты с вакуумным уплотнением, позволяющие положить в тот же самый чемодан еще больше вещей. По сути, Webpack является инструментом, действующим аналогично. Он представляет собой загрузчик и упаковщик модулей, позволяющий сгруппировать файлы вашего приложения в пакеты. Кроме этого, оно позволяет оптимизировать их размеры, чтобы в тот же самый пакет помещалось больше кода.

Например, для развертывания можно подготовить два пакета: все ваши файлы приложения сводятся в один пакет, а все требуемые среды и библиотеки сторонних

разработчиков — в другой. Как показано на рис. 10.2, задействуя Webpack, можно подготовить отдельные пакеты для развертывания в двух разных целях: для разработки (режим dev) и реального применения (режим prod). В первом случае пакеты создаются в памяти, а во втором Webpack создаст настоящие файлы на диске.

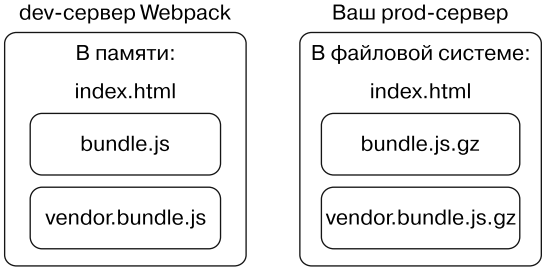


Рис. 10.2. Средства развертывания dev и prod

Приложение удобно создавать в виде набора небольших модулей и сохранять в одном файле один модуль, но для развертывания требуется инструментальное средство для упаковки всех этих файлов в небольшое количество пакетов. Данному средству должно быть известно, как производить сборку дерева зависимостей модуля, избавляя вас от необходимости самому определять порядок загружаемых модулей. Webpack является именно таким средством, и согласно его философии все может быть модулем, включая CSS, изображения и HTML.

Процесс развертывания с помощью Webpack состоит из двух основных этапов.

1. Сборка пакетов (сюда может включаться оптимизация кода).
2. Копирование пакетов на нужный сервер.

Webpack распространяется в виде npm-пакета, и, как и все остальные инструментальные средства, его можно устанавливать либо глобально, либо локально. Начнем с глобальной установки:

```
npm install webpack -g
```

ПРИМЕЧАНИЕ

В данной главе используется Webpack 2.1.0, который на момент написания этих строк представлял собой бета-выпуск. Для его глобальной установки мы задействовали команду npm i webpack@2.1.0-beta.25 -g.

Чуть позже Webpack будет установлен локально путем добавления его в раздел `devDependencies` файла `package.json`. Но его глобальная установка позволит быстро увидеть процесс превращения приложения в пакет.

СОВЕТ

Рекомендованный список ресурсов Webpack (документация, видео, библиотеки и т. д.) можно найти на GitHub. Обратите внимание на awesome-webpack: https://github.com/webpack-contrib/awesome-webpack.

10.1.1. Hello World с Webpack

Познакомимся с Webpack через самый простой пример Hello World, состоящий из двух файлов — `index.html` и `main.js`. Файл `index.html` имеет следующий вид (листинг 10.1).

Листинг 10.1. Содержимое файла index.html

```
<!DOCTYPE html>
<html>
<body>
  <script src="main.js"></script>
</body>
</html>
```

Файл `main.js` гораздо короче (листинг 10.2).

Листинг 10.2. Содержимое файла main.js

```
document.write('Hello World!');
```

Откройте в каталоге, содержащем этот файл, окно командной строки и запустите следующую команду:

```
webpack main.js bundle.js
```

Файл `main.js` является исходным, а файл `bundle.js` в том же самом каталоге — выходным. Обычно в имя выходного файла включается слово *bundle*. Результат запуска предыдущей команды показан на рис. 10.3.

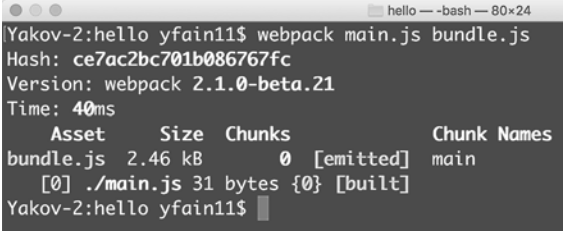


Рис. 10.3. Создание первого пакета

Обратите внимание: размер созданного файла `bundle.js` больше размера файла `main.js`, поскольку Webpack не просто копирует один файл в другой, но и добавляет другой код, требуемый этим пакетом. Создание пакета из одного файла вряд ли принесет какую-то пользу, поскольку тем самым увеличивается размер файла, а вот для приложения, состоящего из множества файлов, упаковка всех файлов в единый пакет имеет смысл. Вы сможете в этом убедиться, прочитав данную главу.

Теперь нужно внести изменения в тег `<script>` в файле HTML, чтобы включить `bundle.js` вместо `main.js`. Это крохотное приложение выведет то же самое сообщение Hello World!, что и его первоисточник.

Webpack позволяет указывать в командной строке разные ключи, но лучше все же сконфигурировать процесс упаковки, выполняемой Webpack, в файле `webpack.config.js`, являющемся файлом JavaScript. Простой файл конфигурации имеет следующий вид (листинг 10.3).

Листинг 10.3. Содержимое файла webpack.config.js

```
const path = require('path');
module.exports = {
  entry: "./main",
  output: {
    path: './dist',
    filename: 'bundle.js'
  }
};
```

Для создания пакета Webpack должен знать об основном модуле *(точке входа)* вашего приложения, который может иметь зависимости от других модулей или библиотек сторонних производителей (другие точки входа). В исходном состоянии упаковщик добавляет к имени входной точки, указанной в свойстве `entry`, расширение `.js`. Webpack загружает модуль входной точки и выстраивает в памяти дерево из всех модулей-зависимостей. Прочитав конфигурационный файл, показанный в листинге 10.3, Webpack будет знать, что вход в приложение расположен в файле `./main.js` и получившийся в результате работы этого средства файл `bundle.js` должен быть сохранен в каталоге `./dist`, имя которого часто используется для распространяемых пактов.

СОВЕТ

Сохранение выходных файлов в отдельном каталоге позволит настроить свою систему контроля версиями на исключение созданных файлов. Если применяется система контроля версиями Git, то добавьте каталог dist в файл .gitignore.

Можно указать более одной точки входа, предоставив в качестве значения свойства `entry` массив:

```
entry: ["./other-module", "./main"]
```

В таком случае в начале работы будет загружен каждый из этих модулей.

ПРИМЕЧАНИЕ

Для создания нескольких пакетов значения для свойства entry нужно указать в виде объектов, а не строк. Соответствующий пример, в котором код Angular будет помещен в один пакет, а код приложения в другой, мы приведем в этой главе чуть позже.

Если файл конфигурации Webpack находится в текущем каталоге, то предоставлять какие-либо параметры командной строки не нужно, и для создания

пакетов можно просто воспользоваться командой `webpack`. Другой вариант предусматривает запуск упаковщика в режиме отслеживания с помощью ключа командной строки `--watch` или `-w`, чтобы при любом изменении файлов приложения Webpack выполнял автоматическую пересборку пакета:

```
webpack --watch
```

Можно также заставить Webpack запускать режим отслеживания путем добавления в `webpack.config.js` следующей записи:

```
watch: true
```

Использование webpack-dev-server

В предыдущих главах для обслуживания ваших приложений использовался live-сервер, но Webpack поставляется со своим собственным webpack-dev-сервером, который должен устанавливаться отдельно. Обычно этот упаковщик добавляют к существующему npm-проекту и устанавливают Webpack и его разработочный сервер локально путем запуска следующей команды:

```
npm install webpack webpack-dev-server --save-dev
```

Данная команда приведет к установке всех требуемых файлов в подкаталог `node_modules` и к добавлению `webpack` и `webpack-dev-server` к разделу `devDependencies` файла `package.json`.

Следующая версия Hello World находится в каталоге `hello-world-devserver` и включает показанный далее файл `index.html` (листинг 10.4).

Листинг 10.4. Содержимое файла hello-world-devserver/index.html

```
<!DOCTYPE html>
<html>
<body>
  <script src="/bundle.js"></script>
</body>
</html>
```

JavaScript-файл `main.js` остается прежним:

```
document.write('Hello World!');
```

Файл `package.json` в проекте `hello-world-devserver` выглядит следующим образом (листинг 10.5).

Листинг 10.5. Содержимое файла hello-world-devserver/package.json

```
{
  "name": "first-project",
  "version": "1.0.0",
  "description": "",
  "main": "main.js",
  "scripts": {
    "start": "webpack-dev-server"
```

```
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "webpack": "^2.1.0-beta.25",
    "webpack-dev-server": "^2.1.0-beta.0"
  }
}
```

Обратите внимание: команда `npm start` настроена на запуск локального webpack-dev-сервера.

ПРИМЕЧАНИЕ

Когда приложение обслуживается webpack-dev-сервером, он запустится на порте по умолчанию 8080 и станет создавать пакеты в памяти без сохранения их в файле. После этого webpack-dev-сервер станет заново компилировать и обслуживать новые версии пакетов после каждого внесения изменений в код.

Вы можете добавить раздел конфигурации webpack-dev-сервера в раздел `devServer` файла `webpack.config.js`. Там можно поместить любые ключи, допускаемые webpack-dev-сервером в командной строке (их описание можно найти в документации по Webpack на http://webpack.github.io/docs/webpack-dev-server.html). Файлы, которые должны обслуживаться из текущего каталога, можно указать следующим образом:

```
devServer: {
  contentBase: '.'
}
```

Ниже показан весь конфигурационный файл для проекта `hello-world-devserver` (листинг 10.6), который может быть повторно использован обеими командами: как `webpack`, так и `webpack-dev-server`.

Листинг 10.6. Содержимое файла hello-world-devserver/webpack.config.js

```
const path = require('path');
module.exports = {
  entry: {
    'main': './main.js'
  },
  output: {
    path: './dist',
    filename: 'bundle.js'
  },
  watch: true,
  devServer: {
    contentBase: '.'
  }
};
```

В листинге 10.6 два ключа нужны, только если вы планируете запустить команду `webpack` в отслеживающем режиме и настроить ее на создание выходных файлов на диске:

- Node-модуль `path` разрешает в вашем проекте относительные пути (в данном случае в нем указывается каталог `./dist`);
- `watch: true` запускает Webpack в отслеживающем режиме.

Если дать команду `webpack-dev-server`, то предыдущие два ключа не используются. Webpack-dev-сервер всегда запускается в отслеживающем режиме, не выводит файлы на диск и собирает пакеты в памяти. Свойство `contentBase` позволяет webpack-dev-серверу узнать местоположение вашего файла `index.html`.

Попробуем запустить приложение Hello World путем обслуживания приложения web-pack-dev-сервером. Для запуска этого сервера введите в окне команд команду `npm start`. На консоли webpack-dev-сервер станет регистрировать вывод, начинающийся с URL, который можно использовать с применением браузера, и по умолчанию имеющий вид http://localhost:8080.

Откройте в своем браузере этот веб-адрес и увидите окно, в котором выведено сообщение Hello World. Измените текст в `main.js`: Webpack автоматически пересоберет пакет, и сервер перезагрузит свежее содержимое.

Разрешение имен файлов

Все это хорошо, но вы пишете код в TypeScript, следовательно, Webpack нужно уведомить, что ваши модули могут быть не только в файлах с расширением `.js`, но и в файлах `.ts`. В файле `webpack.config.js`, показанном в листинге 10.6, имя указано с расширением: `main.js`. Но можно указывать только имена файлов без каких-либо расширений, поскольку в файле `webpack.config.js` имеется раздел `resolve`. Как указать упаковщику, что ваши модули могут находиться в файлах с расширениями `.js` или `.ts`, показано в следующем фрагменте кода:

```
resolve: {
  extensions: ['.js', '.ts']
}
```

Файлы TypeScript также должны пройти предварительную обработку (транспиляцию). Вам нужно сообщить Webpack о необходимости перед созданием пакетов транспилировать файлы вашего приложения с расширением `.ts` в файлы с расширением `.js`; как это делается, будет показано ниже.

Обычно средства автоматизации сборок предоставляются со способом указания дополнительных задач, которые необходимо выполнить в ходе процесса сборки. Webpack предлагает загрузчики и дополнительные модули, позволяющие настроить сборки под свои нужды.

10.1.2. Как использовать загрузчики

Загрузчики являются преобразователями, получающими на входе исходный файл и производящими на выходе другой файл (в памяти или на диске), при этом они одновременно ведут работу только с одним файлом. Загрузчик представляет

собой небольшой модуль JavaScript с экспортируемой функцией, совершающей соответствующее преобразование. Например, загрузчик json-loader получает входной файл и проводит его синтаксический анализ, рассчитанный на формат JSON. Загрузчик base64-loader преобразует свой вход в строку, закодированную на основе алгоритма base64. Загрузчики играют роль, сходную с ролью *задач*, выполняемых в других средствах сборки. Некоторые загрузчики входят в состав Webpack, и их не нужно устанавливать отдельно; другие могут быть установлены из открытых репозиториев. В документации по Webpack на GitHub (http://mng.bz/U0Yv) можно найти перечень загрузчиков, а также посмотреть, как установить и использовать те, что нужны вам.

По сути, загрузчик — это функция, написанная на Node-совместимом JavaScript. Чтобы воспользоваться загрузчиком, не включенным в дистрибутив Webpack, нужно его установить, задействуя npm, и включить его в файл `package.json` вашего проекта. Можно либо добавить нужный загрузчик вручную в раздел `devDependencies` файла `package.json`, либо запустить команду `npm install` с ключом `--save-dev`. В случае применения загрузчика ts-loader команда должна иметь следующий вид:

```
npm install ts-loader --save-dev
```

Загрузчики перечислены в файле `webpack.config.js` в разделе `module`. Например, ts-loader можно добавить следующим образом:

```
module: {
  loaders: [
    {
      test: /\.ts$/,
      exclude: /node_modules/,
      loader: 'ts-loader'
    },
  ]
}
```

Эта конфигурация предписывает Webpack проверять (тестировать) каждое имя файла и, если оно соответствует регулярному выражению `\.ts$`, выполнять его предварительную обработку с помощью загрузчика ts-loader. В синтаксисе регулярных выражений знак доллара в конце является признаком того, что вас интересуют только файлы, чьи имена заканчиваются на `.ts`. Поскольку включать в пакет файлы Angular с таким расширением вам не нужно, каталог `node_modules` исключается. Ссылаться на загрузчики можно либо по их полным именам (например, ts-loader), либо по сокращенным, опуская суффикс -loader (например, ts). Если в вашем шаблоне применяются относительные пути (например, `template: "./home.html"`), то нужно воспользоваться загрузчиком angular2-template-loader.

ПРИМЕЧАНИЕ

Загрузчик SystemJS ни в одном из проектов, представленных в данной главе, не используется. Webpack загружает и трансформирует все файлы проекта с помощью одного или нескольких загрузчиков, сконфигурированных на основе типа файла в webpack.config.js.

Использование загрузчиков для файлов HTML и CSS

В предыдущих главах компоненты Angular, хранящие HTML и CSS в отдельных файлах, были указаны в аннотации `@Component` в виде `templateUrl` и `styleUrls` соответственно. Рассмотрим пример:

```
@Component({
  selector: 'my-home',
  styleUrls: ['app/components/home.css')],
  templateUrl: 'app/components/home.html'
})
```

Обычно HTML- и CSS-файлы хранятся в том же каталоге, где расположен код компонента. Можно ли указать путь относительно текущего каталога?

Webpack позволяет сделать это:

```
@Component({
  selector: 'my-home',
  styles: [home.css'],
  templateUrl: 'home.html'
})
```

При создании пакетов Webpack автоматически добавляет инструкции `require()` для загрузки файлов CSS и HTML, заменяя предыдущий код следующим:

```
@Component({
  selector: 'my-home',
  styles: [require('./home.css')],
  templateUrl: require('./home.html')
})
```

Затем он проверяет каждую инструкцию `require()` и заменяет ее содержимым требуемого файла, применяя загрузчики, указанные для соответствующих типов файлов. Используемая здесь инструкция `require()` не похожа на такую же инструкцию из CommonJS: это внутренняя функция упаковщика, оповещающая его о том, что указанные файлы являются зависимостями. Функция `require()` не только загружает файлы, но и может перезагрузить их при изменениях (если запустить данное средство в режиме отслеживания или применить webpack-dev-сервер).

Относительные пути в шаблонах при использовании SystemJS

Хорошо, что в Webpack поддерживаются относительные пути. Но как быть, если нужно иметь возможность загружать одно и то же приложение с помощью либо SystemJS, либо Webpack?

По умолчанию в Angular для внешних файлов должны применяться полные пути, начинающиеся с корневого каталога приложения. Решение переместить компонент в другой каталог может потребовать внесения изменений в код.

Но если используется SystemJS и код компонента и его файлов HTML и CSS хранится в одном и том же каталоге, то можно задействовать специальное свойство `moduleId`. После присвоения этому свойству специальной привязки `__moduleName` SystemJS

загрузит файлы, имеющие отношение к текущему модулю, без необходимости указания полного пути:

```
declare var __moduleName: string;
@Component({
  selector: 'my-home',
  moduleId:__moduleName,
  templateUrl: './home.html',
  styleUrls: ['./home.css']
})
```

Дополнительные сведения об относительных путях можно найти в документации по Angular в разделе Component-Relative Paths на `http://mng.bz/47w0`.

В dev-режиме для обработки HTML вы будете использовать загрузчик raw-loader, выполняющий преобразование файлов с расширением `.html` в строки. Чтобы установить этот загрузчик и сохранить его в разделе `devDependencies` файла `package.json`, запустите следующую команду:

```
npm install raw-loader --save-dev
```

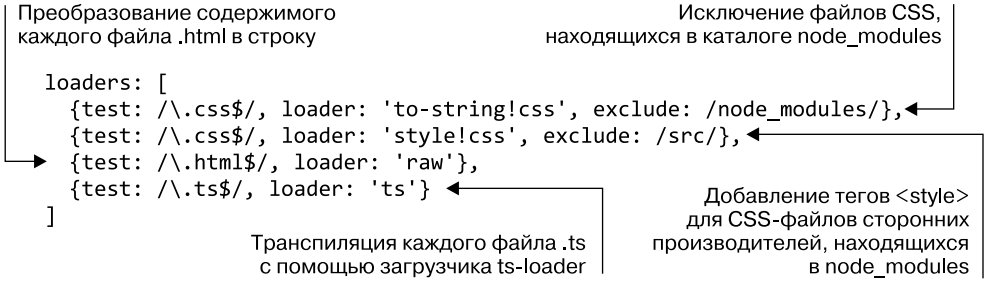
В prod-режиме вы будете применять загрузчик html-loader, удаляющий из HTML-файлов лишние пробелы, символы новой строки и комментарии:

```
npm install html-loader --save-dev
```

Для обработки CSS вы задействуете загрузчики css-loader и style-loader; все связанные CSS-файлы будут встроены в процессе сборки. Загрузчик css-loader проводит синтаксический разбор CSS-файлов и сводит количество стилей к минимуму. Загрузчик style-loader встраивает CSS в виде тега `<style>` в страницу в динамическом режиме в ходе выполнения кода. Чтобы установить эти загрузчики и сохранить их в разделе `devDependencies` файла `package.json`, запустите следующую команду:

```
npm install CSS-loader style-loader --save-dev
```

Загрузчики могут быть выстроены в цепочку с помощью символа конвейера. Следующий фрагмент кода взят из файла `webpack-config.js`, который включает массив загрузчиков. Когда загрузчики указаны в виде массива, они выполняются снизу вверх (то есть загрузчик `ts` в этом примере будет выполнен первым). Фрагмент, приведенный ниже, взят из учебного проекта, рассматриваемого в следующем разделе, где файлы CSS расположены в двух папках (`src` и `node_modules`):



Сначала исключаются CSS-файлы, находящиеся в каталоге `node_modules`, чтобы это преобразование применялось только к элементам приложения. Здесь выстроены в цепочку загрузчики to-string и css. Загрузчик css выполняется первым, превращая CSS в модуль JavaScript, а затем его выход передается по конвейеру загрузчику to-string для извлечения строки из созданного кода JavaScript. Получившаяся строка встраивается в соответствующие компоненты в аннотации `@Component` вместо `require()`, чтобы Angular мог применить верную стратегию `ViewEncapsulation`.

Затем нужно, чтобы Webpack встроил сторонние файлы CSS, находящиеся в каталоге `node_modules` (а не в каталоге `src`). Загрузчик css-loader считывает CSS, создает модуль JavaScript и передает его загрузчику style-loader, который генерирует теги `<style>` с загруженным кодом CSS и встраивает их в раздел `<head>` документа HTML. И наконец, файлы HTML превращаются в строки, и транспилируется код TypeScript.

СОВЕТ

> В Angular нужно, чтобы код CSS был инкапсулирован в компоненты для получения преимуществ ViewEncapsulation, в соответствии с объяснениями, приведенными в главе 6. Именно поэтому код CSS встраивается в код JavaScript. Но есть способ сборки отдельного пакета, содержащего только CSS, путем использования дополнительного модуля ExtractTextPlugin. Если применяется препроцессор CSS, нужно установить и задействовать загрузчик sass-loader или less-loader.

Для чего нужны предзагрузчики и постзагрузчики

Иногда непосредственно перед тем, как загрузчик начнет свои преобразования, требуется выполнить дополнительную обработку файла. Например, может понадобиться прогнать ваши TypeScript-файлы через утилиту TSLint, чтобы проверить код на читаемость, обслуживаемость и наличие функциональных ошибок. Для этого к конфигурационному файлу Webpack нужно добавить раздел `preLoaders`:

```
preLoaders: [
  {
    test: /\.ts$/,
    exclude: /node_modules/,
    loader: "tslint"
  }
]
```

Предзагрузчики всегда запускаются перед загрузчиками, и если при их работе произойдут какие-либо ошибки, то отчет о них выводится в командную строку. Можно также настроить постобработку путем добавления в файл `webpack.config.js` раздела `postLoaders`.

10.1.3. Как использовать дополнительные модули

Если загрузчики Webpack преобразуют файлы поочередно, то дополнительные модули имеют доступ ко всем файлам и могут обрабатывать их до или после запуска загрузчиков. Например, дополнительный модуль CommonsChunkPlugin позволяет создать отдельный пакет для общих модулей, которые требуются различным сценариям в вашем приложении. Дополнительный модуль CopyWebpackPlugin может копировать в каталог сборки `build` либо отдельные файлы, либо целые каталоги. Дополнительный модуль UglifyJSPlugin выполняет минификацию кода всех транспилированных файлов.

Предположим, что вам нужно разбить код приложения на два пакета, `main` и `admin`, и каждый из этих модулей задействует среду Angular. Если только указать две точки входа (`main` и `admin`), то каждый пакет будет включать и код приложения, и свою собственную копию Angular. Во избежание такой ситуации код можно обработать дополнительным модулем CommonsChunkPlugin. Тогда средство Webpack не станет включать код Angular в пакеты `main` и `admin`; оно создаст отдельный совместно используемый пакет, содержащий только код Angular. Тем самым общий размер вашего приложения будет уменьшен, поскольку в него будет включена только одна копия Angular, совместно применяемая двумя модулями приложения. В этом случае файл HTML должен включать сначала пакет поставщика, а затем пакет приложения.

Дополнительный модуль UglifyJSPlugin является оболочкой для минимизатора UglifyJS, который получает код JavaScript и выполняет различные оптимизации. Например, сжимает код, объединяя последовательно указанные инструкции `var`; удаляет неиспользуемые переменные и код, к которому отсутствуют обращения, а также оптимизирует инструкции `if`. Имеющаяся в нем утилита mangler переименовывает локальные переменные, давая им однобуквенные имена. Полное описание UglifyJS можно найти на странице GitHub данного модуля (https://github.com/mishoo/UglifyJS). Эти и другие дополнительные модули будут показаны в следующих разделах.

10.2. Создание базовой конфигурации Webpack для Angular

Мы обсудили основы Webpack, а теперь посмотрим, как упаковать простое Angular-приложение, написанное на TypeScript. Мы создали небольшое приложение, состоящее из компонентов `Home` и `About` и не использующее внешние шаблоны или CSS-файлы. Этот проект находится в каталоге `basic-webpack-starter`, а его структура показана на рис. 10.4.

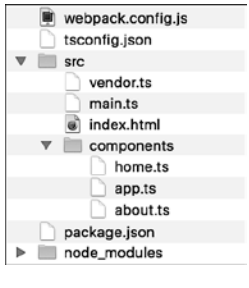


Рис. 10.4. Проект basic-webpack-starter

Сценарий `main.ts` выполняет начальную загрузку компонентов `AppModule` и `MyApp`, настраивающих маршрутизатор, и имеет две ссылки для навигации — либо к `HomeComponent`, либо к `AboutComponent`. Каждый из этих элементов выводит простое сообщение — в контексте данной главы их

фактические функциональные возможности не имеют значения. Наше внимание будет сконцентрировано на создании двух пакетов: одного для среды Angular и ее зависимостей и другого — для кода приложения.

Размер у файла `vendor.ts` небольшой, в нем просто используются операторы `import`, необходимые среде Angular. Это сделано для создания ситуации с двумя точками входа (`main.ts` и `vendor.ts`), содержащими общий код Angular, который будет помещен в отдельный пакет (листинг 10.7).

Листинг 10.7. Содержимое файла basic-webpack-starter/vendor.ts

```
import 'zone.js/dist/zone';
import 'reflect-metadata/Reflect.js';
import '@angular/http';
import '@angular/router';
import '@angular/core';
import '@angular/common';
```

Поскольку эти операторы `import` могут также использоваться в `main.ts`, будет задействован дополнительный модуль CommonsChunkPlugin, чтобы избежать включения кода Angular в оба пакета. Вместо такого включения будет создан отдельный пакет Angular, совместно применяемый основной точкой входа и любой другой точкой входа, если будет принято решение разбить код приложения на мелкие фрагменты.

ПРИМЕЧАНИЕ

Файл vendor.ts должен импортировать все модули, которые нужно включить в общий пакет и убрать из кода приложения. Webpack встроит в общий пакет весь код, необходимый для импортируемых модулей.

Ниже показано содержимое конфигурационного файла `webpack.config.js` (листинг 10.8).

Листинг 10.8. Содержимое файла basic-webpack-starter/webpack.config.js

Инструкция require используется для получения модулей path с целью разрешения путей к файлам. Затем понадобятся два дополнительных модуля: CommonsChunkPlugin и copy-webpack-plugin

```
const path                = require('path');
const CommonsChunkPlugin = require('webpack/lib/optimize/
➥ CommonsChunkPlugin');
const CopyWebpackPlugin  = require('copy-webpack-plugin');

module.exports = {
  entry: {
    'main' : './src/main.ts',

    'vendor': './src/vendor.ts'
  },
  output: {
    path : './dist',
    filename: 'bundle.js'
```

Значение для свойства entry указано в виде объекта, тем самым средству Webpack предписывается создание двух пакетов: одного для точки входа main.ts и другого — для vendor.ts

Получаемые на выходе пакеты будут сохранены в каталоге dist, и имя основной точки входа будет помещено в файл bundle.js. Выход для второй точки входа будет сконфигурирован в разделе plugins

```
  },
  plugins: [
    new CommonsChunkPlugin({ name: 'vendor', filename: 'vendor.bundle.js' }),
    new CopyWebpackPlugin([{from: './src/index.html', to: 'index.html'}])
  ],
  resolve: {
    extensions: ['', '.ts', '.js']
  },
  module: {
    loaders: [
      {test: /\.ts$/, loader: 'ts-loader'}
    ],
    noParse: [path.join(__dirname, 'node_modules', 'angular2', 'bundles')]
  },
  devServer: {
    contentBase: 'src',
    historyApiFallback: true
  },
  devtool: 'source-map'
};
```

Предписывает Webpack создать общий пакет vendor.bundle.js с содержимым, которое может совместно использоваться всеми пакетами приложения

Копирование файла index.html в каталог dist

Для ускорения сборки синтаксический анализ минифицированных файлов Angular, находящихся в каталоге node_modules/angular2/bundles, не проводится

Оповещение dev-сервера о том, что код приложения находится в каталоге src

Создание отображений на исходный код

Поскольку модуль CommonsChunkPlugin поставляется вместе с Webpack, в отдельной установке он не нуждается. После установки дополнительного модуля copy-webpack-plugin упаковщик найдет его в каталоге `node_modules`.

В коде, показанном в листинге 10.8, имеются две точки входа: `main.ts`, содержащая код приложения плюс Angular, и `vendor.ts`, в которой имеется только код Angular. Следовательно, код Angular является общим для обеих точек входа, и этот дополнительный модуль станет извлекать его из `main.ts` и сохранять только в `vendor .bundle.js`.

Хотя для развертывания было бы неплохо упаковать весь код вашего приложения в файл JavaScript, отладку легче выполнять, имея код, который находится в исходном состоянии, в виде отдельных файлов. Добавляя `source-map` к `webpack.config.js`, вы предписываете Webpack создать отображения на исходный код, чтобы можно было увидеть источники файлов JavaScript, CSS и HTML, даже если браузер выполняет код из файла пакета `bundle.js`.

В файле `tsconfig.json` используется режим `"sourceMap": true`, позволяющий создавать отображения на исходный код TypeScript. Браузеры загружают файлы отображения на исходный код, только если на них открыта консоль инструментов разработчика, так что создание отображений на исходный код пригодится даже для развертывания коммерческой сборки. Следует иметь в виду: транспиляция кода будет выполняться загрузчиком ts-loader, поэтому создание кода компилятором tsc можно выключить, установив `"noEmit": true` в файле `tsconfig.json`.

Теперь посмотрим, как будет изменен npm-файл `package.json` с целью включения содержимого, связанного с Webpack. В базовую версию `package.json` в раздел `scripts` будут добавлены две строки.

Раздел devDependencies будет включать webpack, webpack-dev-server, а также требуемые загрузчики и дополнительные модули (листинг 10.9).

Листинг 10.9. Содержимое файла basic-webpack-starter/package.json

```
{
  "name": "basic-webpack-starter",
  "version": "1.0.0",
  "description": "A basic Webpack-based starter project for an Angular 2
  ➥ application",
  "homepage": "https://www.manning.com/books/angular-2-development-with-
  ➥ typescript",
  "private": true,
  "scripts": {
    "build": "webpack",
    "start": "webpack-dev-server --inline --progress --port 8080"
  },

  "dependencies": {
    "@angular/common": "^2.1.0",
    "@angular/compiler": "^2.1.0",
    "@angular/core": "^2.1.0",
    "@angular/http": "^2.1.0",
    "@angular/platform-browser": "^2.1.0",
    "@angular/platform-browser-dynamic": "^2.1.0",
    "@angular/router": "^3.0.0",
    "rxjs": "5.0.0-beta.12",
    "systemjs": "^0.19.37",
    "zone.js": "0.6.21"
  },

  "devDependencies": {
    "@types/es6-shim": "0.0.31",
    "copy-webpack-plugin": "^3.0.1",

    "ts-loader": "^0.8.2",
    "typescript": "^2.0.0",

    "webpack": "^2.1.0-beta.25",

    "webpack-dev-server": "^2.1.0-beta.0"
  }
}
```

Указание Webpack записывать пакеты в выходной каталог dist

Создание пакетов в памяти для последующего обслуживания в браузере с использованием webpack-dev-сервера

Установка определений типов для поддержки особенностей ES6. В файл tsconfig.json также будет добавлен режим компилятора "types":["es6-shim"]

Добавление модуля copy-webpack-plugin в качестве зависимости режима разработки. Обратите внимание: вам не нужно добавлять дополнительный модуль CommonsChunkPlugin, так как он поставляется вместе с Webpack

Добавление в качестве зависимости режима разработки загруженного загрузчика ts-loader

Включение в качестве зависимости режима разработки Webpack

Добавление в качестве зависимости режима разработки разработочного сервера Webpack

ПРИМЕЧАНИЕ

Для обработки файлов определения типов мы задействовали NPM-пакеты из пространства имен @types и режим работы компилятора TypeScript @types. Чтобы применить режим работы @types, требуется установка TypeScript 2.0 или более поздней версии.

Сценарием `start`, как и сценарием `build`, используется одна и та же конфигурация Webpack из файла `webpack.config.js`. Посмотрим, чем эти сценарии отличаются друг от друга.

10.2.1. Команда npm run build

После запуска команды `npm run build` окно команд приобретает вид, показанный на рис. 10.5. Приложение имеет объем 2,5 Мбайт и делает для загрузки всего три запроса.

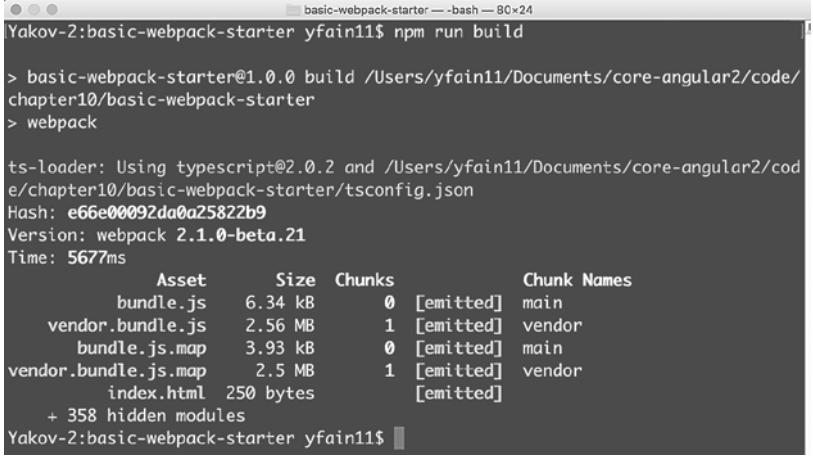


Рис. 10.5. Запуск команды npm run build

Webpack создает два пакета (`bundle.js` и `vendor.bundle.js`) и два соответствующих файла отображения на исходный код (`bundle.js.map` и `vendor.bundle.js.map`), а также копирует `index.html` в выходной каталог `dist`, показанный на рис. 10.6.

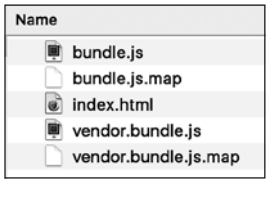


Рис. 10.6. Содержимое каталога dist

В файл `index.html` не входят теги `<script>` для загрузки Angular. Все, что нужно приложению, находится в двух пакетах, включенных в два тега `<script>` (листинг 10.10).

Листинг 10.10. Содержимое файла basic-webpack-starter/index.html

```
<!DOCTYPE html>
<html>
<head>
  <meta charset=UTF-8>
  <title>Basic Webpack Starter</title>
  <base href="/">
</head>
<body>
```

```
  <my-app>Loading...</my-app>
  <script src="vendor.bundle.js"></script>
  <script src="bundle.js"></script>
</body>
</html>
```

Можно открыть окно команд в каталоге `dist` и запустить уже известный вам сервер live-сервер для наблюдения этого простого приложения в работе. Снимок экрана, показанный на рис. 10.7, был сделан после остановки приложения на контрольной точке, поставленной в коде `main.ts`, чтобы показать отображение на исходный код в действии. Несмотря на то что браузер выполняет код из пакетов JavaScript, у вас, благодаря созданию отображений на исходный код, по-прежнему имеется возможность отладки кода конкретного модуля TypeScript.

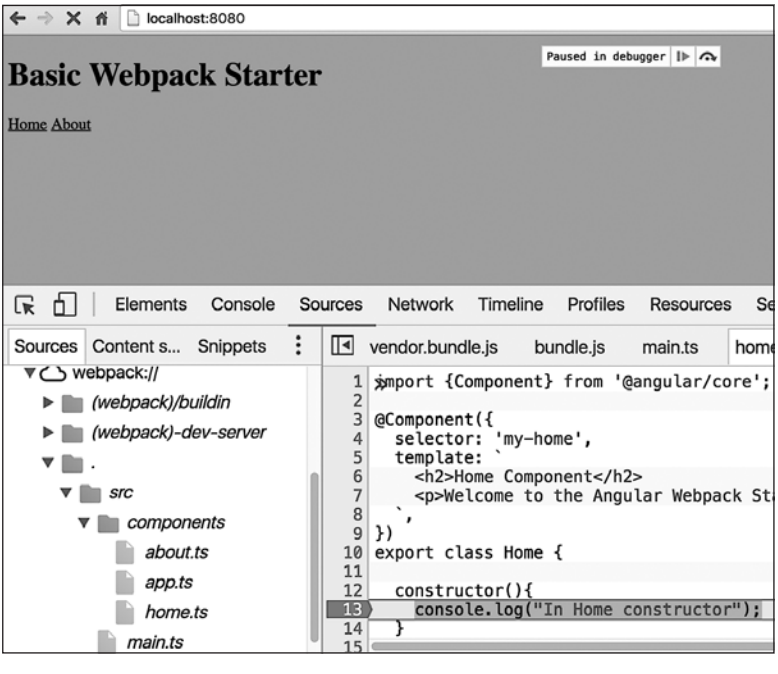


Рис. 10.7. Установка контрольной точки в модуле TypeScript

10.2.2. Команда npm start

Если вместо команды `npm run build` запустить команду `npm start`, то каталог `dist` не будет создан, и webpack-dev-сервер выполнит сборку (включая создание отображения на исходный код), а также обслужит приложение прямо из памяти. Нужно просто открыть в браузере адрес localhost://8080, и ваше приложение обработается. Сейчас оно имеет объем 2,7 Мбайт, но к концу главы станет намного меньше.

Базовый проект Webpack, представленный в этом разделе, хорош для демонстрационных целей. Но настоящим приложениям требуется более серьезная работа с Webpack, и речь о ней пойдет в следующем разделе.

10.3. Создание конфигураций для разработки и для коммерческого применения

В этом разделе будут показаны две версии конфигурационных файлов Webpack (одна для разработки, а другая — для коммерческого применения), которые могут использоваться в качестве отправной точки ваших настоящих Angular-проектов. Весь код, показанный здесь, находится в каталоге `angular2-webpack-starter`. Приложение то же, что и в предыдущем разделе, и состоит из двух компонентов — `Home` и `About`.

В данном проекте в файле `package.json` указано больше npm-сценариев, и в него включены два конфигурационных файла: `webpack.config.js` — для разработочной сборки и `webpack.prod.config.js` — для сборки с расчетом на коммерческое применение.

10.3.1. Конфигурация для разработки

Начнем с файла разработочной конфигурации `webpack.config.js`, который, по сравнению с файлом из предыдущего раздела, получил незначительные дополнения (листинг 10.11). В него добавлен новый дополнительный модуль, DefinePlugin, позволяющий создавать переменные, видимые из кода вашего приложения и доступные для использования средством Webpack в ходе сборки.

Листинг 10.11. Дополненный файл angular2-webpack-starter/webpack.config.js

```
const path               = require('path');
const CommonsChunkPlugin = require('webpack/lib/optimize/
➥ CommonsChunkPlugin');
const CopyWebpackPlugin  = require('copy-webpack-plugin');
const DefinePlugin        = require('webpack/lib/DefinePlugin');

const ENV = process.env.NODE_ENV = 'development';
const HOST = process.env.HOST || 'localhost';
const PORT = process.env.PORT || 8080;

const metadata = {
  env : ENV,
  host: HOST,
port: PORT
};

module.exports = {
  devServer: {
    contentBase: 'src',
    historyApiFallback: true,
    host: metadata.host,
port: metadata.port
},
devtool: 'source-map',
```

Загрузка дополнительного модуля DefinePlugin, позволяющего определять переменные

Node.js использует переменную среды NODE_ENV, значение для которой можно установить в вашем сервере: например, export NODE_ENV=production при работе под управлением Linux

Dev-сервер будет запущен с применением указанного хоста и порта. Константа metadata видна также из кода файла index.html, поэтому, если ваше приложение не развертывается в корневом каталоге вашего веб-сервера, можете определить тут свойство baseURL

```
entry: {
  'main' : './src/main.ts',
  'vendor': './src/vendor.ts'
},
  module: {
    loaders: [
      {test: /\.css$/, loader: 'raw', exclude: /node_modules/},
      {test: /\.css$/, loader: 'style!css?-minimize', exclude: /src/},
      {test: /\.html$/, loader: 'raw'},
      {test: /\.ts$/, loader: 'ts'}
    ],
    noParse: [path.join(__dirname, 'node_modules', 'angular2', 'bundles')]
  },
  output: {
    path : './dist',
    filename: 'bundle.js'
  },
  plugins: [
    new CommonsChunkPlugin({name: 'vendor', filename: 'vendor.bundle.js',
      ➥ minChunks: Infinity}),
    new CopyWebpackPlugin([{from: './src/index.html', to: 'index.html'}]),
    new DefinePlugin({'webpack': {'ENV': JSON.stringify(metadata.env)}})
  ],
  resolve: { extensions: ['.ts', '.js']}
};
```

Загрузчик raw-loader не выполняет преобразование файлов, а встраивает их в качестве строк в шаблоны. Он используется для файлов CSS и HTML

Определение переменной ENV для использования в коде приложения

Переменная `NODE_ENV` используется средой Node.js. Чтобы получить доступ к значению `NODE_ENV` из JavaScript, применяется специальная переменная `process.env.NODE_ENV`. В листинге 10.11 для константы `ENV` устанавливается значение переменной среды `NODE_ENV`, если данная переменная определена, или в противном случае значение `development`. Аналогичным образом задействуются константы `HOST` и `PORT`, и все эти значения будут сохранены в объекте `metadata`.

Переменная `ENV` используется в `main.ts` для вызова Angular-функции `if (webpack.ENV === 'production') enableProdMode();`. Когда режим коммерческой сборки включен, модуль обнаружения изменений, имеющийся в Angular, не выполняет дополнительного прохода с целью удостовериться в отсутствии изменений пользовательского интерфейса в обработчиках жизненного цикла компонентов.

ПРИМЕЧАНИЕ

Даже если в окне команд не устанавливать значение переменной среды Node, у нее имеется значение по умолчанию development, установленное в файле webpack.config.js.

10.3.2. Конфигурация для коммерческого применения

Теперь посмотрим на код файла конфигурации коммерческого применения `webpack.prod.config.js`, который использует следующие дополнительные модули: CompressionPlugin, DedupePlugin, OccurrenceOrderPlugin и UglifyJsPlugin (листинг 10.12).

Листинг 10.12. Содержимое файла angular2-webpack-starter/webpack.prod.config.js

```
const path = require('path');

const CommonsChunkPlugin = require('webpack/lib/optimize/
➥ CommonsChunkPlugin');
const CompressionPlugin = require('compression-webpack-plugin');
const CopyWebpackPlugin = require('copy-webpack-plugin');
const DedupePlugin = require('webpack/lib/optimize/DedupePlugin');
const DefinePlugin = require('webpack/lib/DefinePlugin');
const OccurrenceOrderPlugin = require('webpack/lib/optimize/
➥ OccurrenceOrderPlugin');
const UglifyJsPlugin = require('webpack/lib/optimize/UglifyJsPlugin');

const ENV = process.env.NODE_ENV = 'production';
const metadata = {env: ENV};

module.exports = {
  devtool: 'source-map',
  entry: {
    'main' : './src/main.ts',
    'vendor': './src/vendor.ts'
  },
  module: {
    loaders: [
      {test: /\.css$/, loader: 'to-string!css', exclude: /node_modules/},
      {test: /\.css$/, loader: 'style!css', exclude: /src/},
      {test: /\.html$/, loader: 'html?caseSensitive=true'},
      {test: /\.ts$/, loader: 'ts'}
    ],
    noParse: [path.join(__dirname, 'node_modules', 'angular2', 'bundles')]
  },
  output: {
    path : './dist',
    filename: 'bundle.js'
  },
  plugins: [
    new CommonsChunkPlugin({name: 'vendor', filename: 'vendor.bundle.js',
    ➥ minChunks: Infinity}),
    new CompressionPlugin({regExp: /\.css$|\.html$|\.js$|\.map$/}),
    new CopyWebpackPlugin([{from: './src/index.html', to: 'index.html'}]),
    new DedupePlugin(),
    new DefinePlugin({'webpack': {'ENV': JSON.stringify(metadata.env)}}),

    new OccurrenceOrderPlugin(true),

    new UglifyJsPlugin({
      compress: {screw_ie8 : true},
      mangle: {screw_ie8 : true}
      }
    })
  ],
  resolve: { extensions: ['.ts', '.js']}
};
```

Установка для константы ENV значения переменной среды NODE_ENV, если таковая определена, или "production" в обратном случае

Загрузчик html-loader превращает файл HTML в строку, заменяющую соответствующий вызов require()

Дополнительный модуль CompressionPlugin использует утилиту gzip для подготовки сжатых версий ресурсов, соответствующих этому регулярному выражению

Дополнительный модуль DedupePlugin ведет поиск точно таких же или похожих файлов и исключает дубликаты в своем выводе

В целях оптимизации Webpack заменяет имена модулей числовыми идентификаторами, а дополнительный модуль OccurrenceOrderPlugin дает еще более короткие идентификаторы часто используемым модулям

Дополнительный модуль UglifyJsPlugin минифицирует код JavaScript и применяет утилиту mangler для переименования локальных переменных с присвоением им однобуквенных имен

Процесс сборки будет начинаться с использования команд npm-сценария, включенных в файл `package.json`, в котором содержится больше команд, чем в предыдущем таком же файле, рассмотренном в разделе 10.2. Обратите внимание: в файле `package.json` в команде `build` явно указывается файл `webpack.prod.config.js`, содержащий конфигурацию для коммерческого применения, а команда `start` будет задействовать разработочную конфигурацию из файла `webpack.config.js`, чье имя применяется разработочным сервером упаковщика Webpack (листинг 10.13).

Листинг 10.13. Содержимое файла angular2-webpack-starter/package.json


```
"scripts": {
    "clean": "rimraf dist",

    "prebuild": "npm run clean && npm run test",

    "build": "webpack --config webpack.prod.config.js --progress --profile
    ➥ --colors",

  "start": "webpack-dev-server --inline
  ➥ --progress --port 8080",

  "preserve:dist": "npm run build",

  "serve:dist": "static dist -H '{\"Cache-Control\": \"no-cache,
  ➥ must-revalidate\"}' -z",
"test": "karma start karma.conf.js"

  "prebuild:aot": "npm run clean",
      "build:aot": "ngc -p tsconfig.aot.json && webpack --config
      ➥webpack.prod.config.aot.js --progress --profile --colors",
      "preserve:dist:aot": "npm run build:aot",
      "serve:dist:aot": "static dist -H '{\"Cache-Control\": \"no-cache,
      ➥ must-revalidate\"}' -z"
}
```

Обычно после запуска команды `npm install` используются следующие команды (значение для переменной среды `NODE_ENV` в командной строке устанавливаться не будет).

- `npm start` — запускает разработочный сервер упаковщика Webpack в режиме разработки и обслуживает неоптимизированное приложение. Если открыть в браузере инструменты разработчика, то можно увидеть, что приложение за-

пущено в разработочном режиме, поскольку у переменной ENV имеется значение development, в соответствии с установкой в файле webpack.config.js.

- npm run serve:dist — выполняет команду npm run build для создания оптимизированных пакетов в каталоге dist, а также запускает статический веб-сервер и обслуживает оптимизированную версию приложения. Если открыть в браузере инструменты разработчика, то мы не увидим сообщения о том, что приложение запущено в разработочном режиме, поскольку оно запущено в режиме для коммерческого применения. Значением переменной ENV является production в соответствии с установкой в файле webpack.prod.config.js.
- npm run serve:dist:aot — запускает ngc-компилятор Angular для компиляции перед выполнением (ahead-of-time, AoT), проводимой до создания пакетов. Тем самым устраняется надобность во включении ngc в код приложения и в дальнейшей оптимизации размера пакета.

СОВЕТ

Файлы сценариев разработки и коммерческого применения мы храним в отдельных файлах, несмотря на то что один и тот же файл можно использовать повторно за счет выборочного применения тех или иных разделов конфигурации на основе значения переменной среды. Кто-то определяет два файла и повторно задействует разработочную конфигурацию в конфигурации для коммерческого применения (например, var devWebpackConfig = require('./webpack.config.js');). Исходя из нашего опыта, это затрудняет чтение сценария конфигурации, так что мы храним все сборочные конфигурации целиком в отдельных файлах.

ПРИМЕЧАНИЕ

В каждом из примеров, и в webpack.config.js, и в webpack.prod.config.js, менее 60 строк. Если для подготовки подобной конфигурации сборки предполагается использование Gulp, то она будет включать пару сотен строк кода.

10.3.3. Специальный файл определения типов

Чтобы предотвратить возникновение ошибок компилятора tsc, к приложению нужно добавить специальный файл определения типов. Ошибки компиляции в приложении могут возникать по двум причинам.

- Webpack будет выполнять загрузку и преобразование всех ваших файлов CSS и HTML. В ходе преобразования упаковщик станет заменять все появления кода, похожего на этот:

```
styles: ['home.css'],
template: require('./home.html')
```

преобразованным содержимым:

```
styles: [require('./home.css')],
template: require('./home.html')
```

Это собственная Webpack-функция `require()`, а не такая же функция Node.js. Если запускать компилятор tsc, то предыдущий код станет причиной ошибки компиляции, поскольку компилятор не распознает `require()` с такой сигнатурой.

- Приложение использует константу `ENV`, определенную в конфигурационных файлах Webpack:

```
if (webpack.ENV === 'production') {
  enableProdMode();
}
```

Чтобы компилятор не противился этой переменной, создайте специальный файл определения типов `typings.d.ts` со следующим содержимым (листинг 10.14).

Листинг 10.14. Содержимое файла angular2-webpack-starter/typings.d.ts

```
declare function require(path: string);
declare const webpack: {
  ENV: string
};
```

Определения типов, начинающиеся с ключевого слова `declare`, не имеют прямой ссылки на фактическую реализацию переменных (таких как функция `require()` и константа `ENV`), и в случае изменения кода на вас ложится ответственность за обновление предыдущего файла (например, если вы решите переименовать `ENV` в `ENVIRONMENT`).

На рис. 10.8 показан снимок экрана, который был сделан после запуска команды `npm run serve:dist` и открытия приложения в браузере. Обратите внимание: эта версия приложения делает только три запроса к серверу, и общий размер приложения составляет 180 Кбайт. Оно немного меньше приложения-аукциона, но все же можно сравнить количество запросов к серверу и размер, показанные на рис. 10.1, чтобы увидеть разницу и оценить работу, проделанную Webpack.

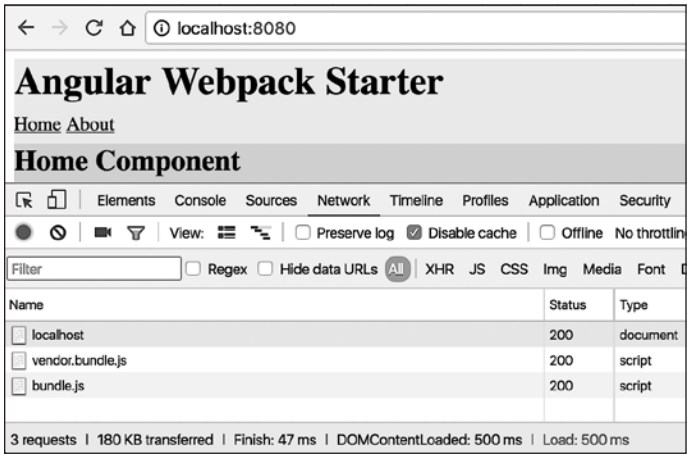


Рис. 10.8. После запуска npm run serve:dist

Теперь посмотрим, как на размер приложения повлияет компиляция перед выполнением. Запустите команду `npm run serve:dist:aot`. Размер приложения стал всего лишь 100 Кбайт! Впечатляет, не правда ли?

ПРИМЕЧАНИЕ

На момент написания этих строк (Angular 2.1.0) AoT-компиляция производила объем меньше, чем JIT, только для небольших приложений.

Организация непрерывности

Выполняя обязанности руководителя проекта, нужно обеспечить автоматизацию основных его процессов. Самая крупная ошибка, которую может допустить руководитель проекта, — дать разрешение какому-нибудь эксперту-разработчику Джону Смиту провести сборку и развертывание приложения вручную. Джон, как человеческое существо, может в один прекрасный день заболеть, уйти в отпуск или даже уволиться. Автоматизация процессов сборки и развертывания служит гарантией непрерывности разработки программного средства. На ранних стадиях разработки вашего проекта должны быть организованы следующие процессы.

- *Непрерывная интеграция* (Continuous integration, CI) — организация процесса, запускающего сценарии сборки несколько раз в день, например после каждого объединения кода в репозитории исходного кода. Сценарии сборки включают модульные тесты, минификацию и упаковку. Вам следует установить и настроить CI-сервер, чтобы обеспечить постоянное пребывание ведущей ветви кода приложения в работоспособном состоянии и чтобы никогда не приходилось отвечать на вопрос: «Кто загубил сборку?»
- *Непрерывная доставка* (Continuous delivery, CD) — процесс, подготавливающий ваше приложение к развертыванию. CD заключается в предложении вашим пользователям дополнительных функциональных возможностей и исправлений.
- *Непрерывное развертывание* (Continuous deployment) — процесс развертывания новой версии приложения, подготовленной при выполнении фазы CD. Позволяет получать от ваших пользователей частые отзывы, гарантирующие, что ваша команда разрабатывает продукт, реально необходимый пользователям.

Фронтенд-разработчики зачастую сотрудничают с командой, создающей серверную сторону приложения. У этой команды уже могут быть организованы процессы CI и CD, и вам следует научиться интегрировать вашу сборку с помощью инструментов, используемых для серверной стороны.

Если вы до сих пор верите, что развертывание вручную не является преступлением, то почитайте о произошедшем с Knight Capital Group, обанкротившейся за 45 минут из-за человеческой ошибки в ходе развертывания. В 2014 г. Дуг Севен (Doug Seven) написал статью Knightmare: A DevOps Cautionary Tale, в которой описан этот инцидент (https://dougseven.com/2014/04/17/knightmare-a-devops-cautionary-tale/). Подводя черту, стоит отметить: процессы сборки и развертывания должны быть автоматизированы и носить характер повторяемости, а также не могут зависеть от одного-единственного представителя технического персонала.

СОВЕТ

При сборке крупного приложения с мегабайтами кода JavaScript может появиться желание разбить код приложения на несколько модулей (точек входа) и превратить каждый из них в пакет. Предположим, у вашего веб-приложения имеется модуль нечасто используемого пользовательского профиля. Удаление кода, реализующего профиль пользователя, приведет к уменьшению начального размера главной страницы вашего приложения, и код профиля пользователя будет загружаться только по необходимости. В популярном веб-приложении Instagram определяется более десятка точек входа.

10.4. Что такое Angular CLI

Изначально попасть в мир Angular-разработки было довольно сложно, поскольку требовалось изучить и вручную сконфигурировать множество инструментов. Даже чтобы приступить к разработке простого приложения, надо было владеть языком TypeScript и уметь им пользоваться, знать компилятор TypeScript, модули ES6, SystemJS, npm и разработочный веб-сервер. Для работы с настоящим проектом требовалось также изучить порядок тестирования и упаковки приложения.

Для резкого старта процесса разработки команда Angular создала утилиту под названием Angular CLI (https://github.com/angular/angular-cli), являющуюся инструментом командной строки, охватывающим все стадии жизненного цикла Angular-приложения, от разработки временной платформы и создания исходного приложения до реализации шаблона для компонентов, модулей, сервисов и т. д. Созданный код также включает предварительно сконфигурированные файлы для модульного тестирования и упаковки с использованием Webpack.

Глобальную установку Angular CLI можно выполнить, написав следующую команду:

```
npm install -g angular-cli
```

Angular CLI и Webpack

Angular CLI усиливается упаковщиком Webpack. Внутренне сгенерированный с помощью CLI, проект включает файлы конфигурации Webpack, подобные рассмотренным в данной главе.

Но даже притом, что внутренне Angular CLI использует Webpack, это не позволяет вам изменять конфигурацию Webpack, что может помешать реализовывать специализированные требования сборки (например, специальную стратегию сборки) с применением CLI. В таком случае сконфигурировать проект можно вручную (без CLI), задействуя знания, почерпнутые в настоящей главе.

10.4.1. Запуск нового проекта с Angular CLI

После установки CLI в вашей переменной среды PATH открывается доступ к выполняемому двоичному файлу ng. Команда ng будет использоваться для вызова команд Angular CLI.

Чтобы создать Angular-приложение, применяется команда new:

```
ng new basic-cli
```

CLI создаст новый каталог basic-cli, в который будут включены все файлы, требуемые для простого приложения. Для его запуска наберите команду ng serve и откройте в окне своего браузера адрес http://localhost:4200. Будет показана страница с выведенным на ней сообщением app works!. CLI автоматически установит все требуемые зависимости и создаст в папке локальный git-репозиторий. Сгенерированная структура проекта показана на рис. 10.9.

Рис. 10.9. Структура CLI-проекта

Вкратце рассмотрим основные файлы и каталоги проекта.

- e2e — каталог для сквозных тестов.
- src/app — основной каталог для кода приложения. Здесь обычно создаются подкаталоги для маршрутов и дочерних компонентов, но CLI не выставляет это обязательным требованием.
- src/assets — все в данном каталоге в процессе сборки должно быть скопировано как есть в каталог dist.
- src/environments — здесь указываются настройки, относящиеся к среде. Можно создать произвольное количество специализированных сред, например контроля качества — QA, обкатки и коммерческого применения.
- angular-cli.json — основной конфигурационный файл Angular CLI. Здесь можно настроить местоположение файлов и каталогов, от которых зависит CLI, например глобальных CSS-файлов и ресурсов.

10.4.2. Команды CLI

В CLI предоставляется ряд команд, пригодных для использования в целях управления Angular-приложением. Наиболее востребованные из них, которые можно применять при разработке и подготовке версии приложения, предназначенной для коммерческого применения, перечислены в табл. 10.1.

Таблица 10.1. Наиболее востребованные команды CLI

Команда	Описание
ng serve	Приводит к сборке пакета в памяти и запускает включенный в среду webpack-dev-сервер, который будет запускаться и пересобирать пакеты после каждого изменения, вносимого в код приложения
ng generate	Приводит к созданию различных итоговых продуктов вашего проекта. Можно также воспользоваться сокращенной версией команды: ng g. Для вывода списка всех доступных ключей нужно ввести команду ng help generate. Примеры: • ng g c <название-компонента> — приводит к созданию для компонента четырех файлов: TypeScript-файла для исходного кода, TypeScript-файла для модульных тестов элемента, HTML-файла для шаблона и CSS-файла для стилевых настроек представления компонента. Если нужно встроить CSS и шаблоны в создаваемый элемент, то эту команду следует запустить с ключами: например, ng g c product --inline-styles --inline-template. При отсутствии надобности создавать файл спецификаций следует воспользоваться ключом --spec=false; • ng g s <название-сервиса> — приводит к созданию двух TypeScript-файлов: одного для исходного кода и второго — для модульных тестов
ng test	Запускает модульные тесты, задействуя средство для запуска тестов Karma
ng build	Приводит к созданию JavaScript-пакетов с транспилированным кодом приложения и всеми встроенными зависимостями. Пакеты сохраняются в каталоге dist

Чтобы выполнить сборку оптимизированной версии приложения, команду `ng build` запускают, как правило, с ключами `-prod --aot`. Без указания этих ключей размер создаваемых пакетов основного генерируемого приложения равен около 2,5 Мбайт. Создание пакетов с помощью команды `ng build -prod` сокращает размер пакета до 800 Кбайт, а также приводит к созданию версии пакета, сжатой средством gzip (190 Кбайт).

Чтобы оптимизировать пакет под развертывание с целью коммерческого применения, настройте компиляцию перед выполнением (ahead-of-time compilation), запустив команду `ng build --prod --aot`. Теперь размер пакета составит примерно 450 Кбайт, а его версия, сжатая средством gzip, уменьшится до 100 Кбайт. На рис. 10.10 показано содержимое каталога `dist` приложения basic-cli после запуска этой команды.

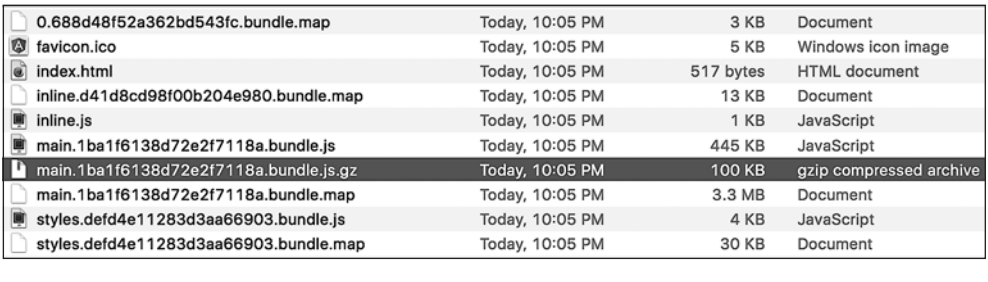


Рис. 10.10. Каталог dist после сборки для коммерческого применения

ПРИМЕЧАНИЕ

Чтобы увидеть данный пакет размером 100 Кбайт загруженным в браузер, нужно воспользоваться веб-сервером, поддерживающим обслуживание файлов, которые были сжаты средством gzip непосредственно перед сборкой. Можно, к примеру, задействовать статический node-сервер, применявшийся в подразделе 10.3.2.

На этом мы завершаем краткий обзор средства Angular CLI, которое избавляет от необходимости создавать вручную множество конфигурационных файлов и производит коммерческие сборки с высокой степенью оптимизации. Описание всех команд Angular CLI можно найти на https://cli.angular.io/reference.pdf.

Может возникнуть вопрос: «Зачем изучать внутреннее устройство Webpack, если Angular CLI сконфигурирует Webpack за нас?» Мы хотели объяснить, как все это работает, на случай, если вы решите вручную создавать свои сборки и проводить их тонкую настройку без использования Angular CLI. К тому же в будущем Angular CLI может позволить вам вносить изменения в автоматически создаваемые файлы конфигурации Webpack, и было бы неплохо знать о том, как это делается.

СОВЕТ

Для ускорения установки зависимостей проектов Angular CLI воспользуйтесь вместо npm диспетчером пакетов Yarn. Подробности можно найти на https://yarnpkg.com/lang/en/.

10.5. Практикум: развертывание онлайн-аукциона с помощью Webpack

В данном упражнении никакой новый код приложения разрабатываться не будет. Целью ставится использование Webpack для сборки и развертывания оптимизированной версии онлайн-аукциона. В процесс сборки будет также интегрировано средство для запуска тестов Karma.

Для этой главы мы переработали проект аукциона из главы 8, который применяет тот же самый файл `package.json` между клиентской и серверной частями приложения. Теперь клиент и сервер станут отдельными приложениями со своими собственными файлами `package.json`. Раздельное содержание клиентского и серверного кода упрощает автоматизацию процесса сборки.

Angular и безопасность

В Angular имеется встроенная защита против наиболее распространенных уязвимостей и атак веб-приложений. В частности, для предотвращения межсайтовых сценарных атак выполняется блокировка вредоносного кода, препятствующая его входу в DOM-модель. Что касается изображений, злоумышленник может заменить их вредоносным кодом в атрибуте `src` тега `<img>`.

В приложении-аукционе используются изображения, получаемые с сайта https://placeholder.com/, который станет блокироваться в ходе упаковки, если только не будет специально сделана запись о доверии данному сайту. В приложение-аукцион, применяемое в данной главе, добавляется код, утверждающий, что вы доверяете изображениям, получаемым с этого сайта, и их не нужно блокировать.

Именно поэтому в элементы аукциона, которые задействуют изображения, поступающие со стороннего сервера, добавляется код, предотвращающий обезвреживание изображений. Например, конструктор для компонента `ProductItemComponent` выглядит следующим образом:

```
constructor(private sanitizer: DomSanitizer) {
    this.imgHtml = sanitizer.bypassSecurityTrustHtml(`
        <img src="http://placehold.it/320x150">`);
}
```

Подробности мер обеспечения безопасности Angular можно найти в документации на https://angular.io/guide/security.

В файле `package.json` из каталога `client` имеются npm-сценарии, необходимые для коммерческой сборки пакета, а также для запуска webpack-dev-сервера в режиме разработки. В каталоге `server` имеется свой собственный файл `package.json` с npm-сценариями для запуска Node-сервера аукциона, здесь упаковка не нужна. С технической точки зрения у вас имеются два независимых приложения со своими собственными отдельно сконфигурированными зависимостями. Этот проект-практикум начинается с использования исходного кода, который находится в каталоге `auction` главы 10.

10.5.1. Запуск Node-сервера

Серверный файл `package.json` выглядит следующим образом (листинг 10.15).

Листинг 10.15. Содержимое файла auction/package.json

```
{
  "name": "ng2-webpack-starter",
  "description": "Angular 2 Webpack starter project suitable for a production
  ➥ grade application",
  "homepage": "https://www.manning.com/books/angular-2-development-with-
  ➥ typescript",
  "private": true,
  "scripts": {
    "tsc": "tsc",
    "startServer": "node build/auction.js",
    "dev": "nodemon build/auction.js"
  },
  "dependencies": {
    "express": "^4.13.3",
    "ws": "^1.0.1"
  },
```

```
  "devDependencies": {
    "@types/compression": "0.0.29",
    "@types/es6-shim": "0.0.27-alpha",
    "@types/express": "^4.0.28-alpha",
    "@types/ws": "0.0.26-alpha",
    "compression": "^1.6.1",
    "nodemon": "^1.8.1",
    "typescript": "^2.0.0"
  }
}
```

Обратите внимание: здесь определяется сценарий `tsc` в целях гарантировать использование локальной версии TypeScript 2.0, даже если у вас имеется старая версия компилятора, установленная глобально. Перейдите в командной строке в каталог `server` и запустите команду `npm install`, чтобы получить все необходимые зависимости для серверной части приложения.

Для применения локального компилятора нужно запустить команду `npm run tsc`, которая приведет к транспиляции серверного кода и к созданию файлов `auction.js` и `model.js` (и их отображений на исходный код) в каталоге `build`, в соответствии с конфигурацией, указанной в файле `tsconfig.json`. Это код для сервера аукциона.

Запустите сервер путем ввода команды `npm run startServer`. Он выведет в консоли сообщение Listening on 127.0.0.1:8000.

10.5.2. Запуск клиента аукциона

Клиентскую часть аукциона можно запустить либо в разработочном режиме, либо в режиме коммерческой сборки, используя различные команды npm-сценариев. Раздел npm-сценариев клиентского файла `package.json` имеет следующие команды:

```
"scripts": {
  "clean": "rimraf dist",
  "prebuild": "npm run clean && npm run test",
  "build": "webpack --config webpack.prod.config.js --progress --profile",
  "startWebpackDevServer": "webpack-dev-server --inline --progress --
     port 8080",
  "test": "karma start karma.conf.js",
  "predeploy": "npm run build && rimraf ../server/build/public && mkdirp
  ➥ ../server/build/public",
  "deploy": "copyup dist/* ../server/build/public"
}
```

Большинство команд должно быть вам знакомо, поскольку они использовались в файле `webpack.prod.config.js`, показанном в листинге 10.12. Здесь добавлена новая команда `deploy`, в которой применена команда `copyup`, предназначенная для копирования файлов из клиентского каталога `dist` в серверный каталог `build/public`. В данном случае задействована команда `copyup` из npm-пакета `copyfiles` (https://www.npmjs.com/package/copyfiles). Когда дело касается копирования файлов, этот пакет используется для кросс-платформенной совместимости.

Кроме того, добавлена команда `test` для запуска тестов с помощью программы Karma (см. подраздел 10.5.3).

Поскольку здесь имеется команда `predeploy`, она будет автоматически запускаться при каждом вводе команды `npm run deploy`. В свою очередь, команда `predeploy` запустит команду `build`, которая автоматически включит команду `prebuild`. Последняя запустит команды `clean` и `test`, и только после того, как выполнение всех команд успешно завершится, команда `build` приведет к сборке. И наконец, команда `copyup` приведет к копированию пакетов из каталога `dist` в каталог `server/build/public`.

Перед запуском клиентской части аукциона нужно открыть отдельное окно команд, перейти в каталог `client` и запустить команду `npm install`. Затем следует запустить приложение-аукцион в разработочном режиме путем ввода команды `npm run startWebpackDevServer`. Webpack-dev-сервер привяжется к вашему Angular-приложению и начнет отслеживать запросы браузера, отправляемые через порт 8080. Введите в браузер адрес http://localhost:8080 и увидите знакомый пользовательский интерфейс приложения-аукциона.

ПРИМЕЧАНИЕ

> Разработочная сборка выполнена в памяти, и приложение-аукцион доступно на порте 8080, который является портом, сконфигурированным в файле webpack.config.js.

Откройте вкладку Network (Сеть) на панели Developer Tools (Инструменты разработчика) браузера Chrome. Вы увидите следующее: приложение загружает только что собранные пакеты и его размер еще довольно велик.

Проверьте в консоли журнал Webpack и увидите, какие файлы в какой пакет (или *фрагмент*) попали. В данном случае собираются два фрагмента: `bundle.js` и `vendor.js`. На рис. 10.11 показан небольшой фрагмент журнала Webpack, но вы можете увидеть размер каждого файла. Упакованное приложение (`bundle.js`) имеет объем 285 Кбайт, а код поставщика (`vendor.bundle.js`) — 3,81 Мбайт.

```
Version: webpack 2.1.0-beta.21
Time: 25082ms
                          Asset       Size  Chunks             Chunk Names
  f4769f9bdb7466be65088239c12046d1.eot  20.1 kB       [emitted]
 fa2772327f55d8198301fdb8bcfc8158.woff  23.4 kB       [emitted]
448c34a56d699c29117adc64c43affeb.woff2    18 kB       [emitted]
  e18bbf611f2a2e43afc071aa2f4e1512.ttf  45.4 kB       [emitted]
  89889688147bd7575d6327160d64e760.svg   109 kB       [emitted]
                        bundle.js       285 kB       0  [emitted]  main
                     vendor.bundle.js    3.81 MB       1  [emitted]  vendor
                        bundle.js.map    275 kB       0  [emitted]  main
                     vendor.bundle.js.map    4.05 MB       1  [emitted]  vendor
                        index.html  315 bytes       [emitted]
```

Рис. 10.11. Фрагмент вывода на консоль

В верхней части показано несколько файлов шрифтов, которые не попали в пакет, так как в файле `webpack.config.js` указан параметр `limit`, чтобы исключить вставку в пакет крупных шрифтов:

```
{test: /\.woff$/, loader: "url?limit=10000&minetype=application/font-woff"},
{test: /\.woff2$/, loader: "url?limit=10000&minetype=application/font-woff"},
{test: /\.ttf$/, loader: "url?limit=10000&minetype=application/
➥ octet-stream"},
{test: /\.svg$/, loader: "url?limit=10000&minetype=image/svg+xml"},
{test: /\.eot$/, loader: "file"}
```

В последней строке загрузчику file-loader предписывается копировать шрифты с расширением `.eot` в каталог сборки. Если прокрутить вывод в консоли, то можно будет увидеть, что весь код приложения попал в фрагмент `{0}`, а весь код поставщика — в фрагмент `{1}`.

ПРИМЕЧАНИЕ

> В режиме разработки Angular-приложение под Node-сервером не развертывается. Node-сервер запущен с привязкой к порту 8000, а клиент приложения-аукциона получает обслуживание через порт 8080 и обменивается данными с Node-сервером с помощью протоколов HTTP и WebSocket. Развертывание Angular-приложения под Node-сервером будет выполнено на следующем этапе.

Теперь остановите webpack-dev-сервер (но не Node-сервер), нажав сочетание клавиш Ctrl+C, находясь в окне команд, из которого запускался клиент. Запустите *коммерческую сборку,* введя команду `npm run deploy`. Эта команда приведет к подготовке оптимизированной сборки и к копированию ее файлов в каталог `../server/build/public`, к которому принадлежит все статическое содержимое Node-сервера.

Node-сервер перезапускать не нужно, поскольку здесь выполняется развертывание только статического кода. Но, чтобы посмотреть версию аукциона, предназначенную для коммерческого применения, вам нужно воспользоваться портом 8000, на котором запущен Node-сервер.

Откройте в окне браузера адрес http://localhost:8000. Там вы увидите приложение-аукцион, обслуживаемое Node-сервером. Откройте панель Developer Tools (Инструменты разработчика) браузера Chrome на вкладке Network (Сеть) и обновите вывод приложения. Вы увидите, что размер оптимизированного приложения существенно уменьшился. На рис. 10.12 показано: общий размер приложения составляет 349 Кбайт (по сравнению с 5,5 Мбайт в показанной ранее на рис. 10.1 неупакованной версии).

Чтобы загрузить `index.html`, два пакета и черно-белые изображения, представляющие товары, браузер делает к серверу девять запросов. Кроме того, можно увидеть запрос на данные о товарах, который клиент выполнил, используя имеющийся

в Angular Http-запрос. Строка, оканчивающаяся на `.woff2`, является шрифтом, загруженным Twitter-фреймворком Bootstrap.

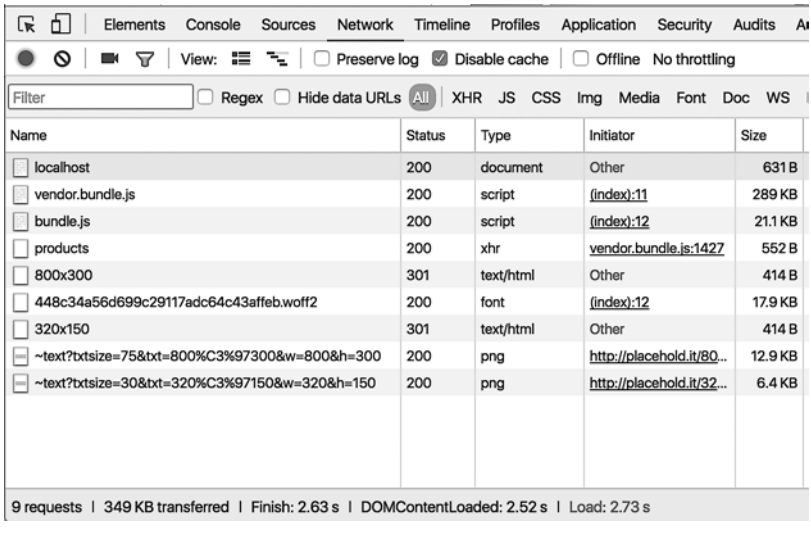


Рис. 10.12. Все, что загружено в prod-версии аукциона

Загрузчик url-loader работает примерно также, как и file-loader, но может встраивать файлы меньше указанного лимита в CSS, где они определены. В качестве лимита для файлов с именами, оканчивающимися на `.woff`, `.woff2`, `.ttf` и `.svg`, был указан размер 10 000 байт. Один из более крупных файлов (17,9 Кбайт), встроен не был.

Каждый шрифт представлен в нескольких форматах, таких как `.eot`, `.woff`, `.woff2`, `.ttf` и `.svg`. Для работы со шрифтами существует несколько невзаимоисключающих вариантов:

- встроить их все в пакет и позволить браузеру выбрать для использования один из них;
- встроить формат шрифта, поддерживаемый самым старым браузером, на поддержку которого рассчитано ваше приложение. Это значит, что более новым браузерам придется загружать файлы в два или три раза больше, чем им нужно;
- не встраивать никакие шрифты и позволить браузеру выбрать и загрузить один из шрифтов, пользующийся наилучшей поддержкой.

Принятая здесь стратегия заключалась во встраивании только избранных шрифтов, отвечающих определенным критериям, и копировании всех остальных шрифтов в папку сборки, что может рассматриваться в качестве сочетания первого и последнего вариантов.

10.5.3. Запуск тестов с помощью Karma

В главе 9 были разработаны три спецификации для модульного тестирования: `ApplicationComponent`, `StarsComponent` и `ProductService`. Здесь будут повторно использоваться те же самые спецификации, но запускаться они будут с применением Karma в качестве составной части процесса сборки.

Поскольку теперь клиент и сервер являются npm-проектами, файлы конфигурации Karma `karma.conf.js` и `karma-test-runner.js` расположены в клиентском каталоге (листинг 10.16).

Листинг 10.16. Содержимое файла auction/client/karma.conf.js

```
module.exports = function (config) {
  config.set({
    browsers : ['Chrome', 'Firefox'],
    frameworks : ['jasmine'],
    reporters : ['mocha'],
    singleRun : true,
    preprocessors: {'./karma-test-runner.js': ['webpack']},
    files : [{pattern: './karma-test-runner.js', watched: false}],
    webpack : require('./webpack.test.config.js'),
    webpackServer: {noInfo: true}
  });
};
```

В главе 9 для демонстрации хода процесса использовались точки, но здесь применяется более наглядный комментатор mocha, который выводит не точки, а сообщения из спецификаций (рис. 10.13)

Указание на то, когда сценарий запуска Karma-тестов должен быть предварительно обработан дополнительным модулем karma-webpack

Загрузка тестовых спецификаций, включенных в файл karma-test-runner.js

Указание на конфигурационный файл Webpack, который должен использоваться средством Karma

Выключение регистрационных записей Webpack, чтобы сообщения этого средства не засоряли вывод средства Karma

Файл `karma.conf.js` намного короче, чем такой же файл из главы 9, поскольку вам не нужно больше конфигурировать файлы для SystemJS, — теперь загрузчиком является Webpack, и файлы уже сконфигурированы в `webpack.test.config.js`. В листинге 10.17 показан код сценария `karma-test-runner.js`, используемого для запуска Karma.

Листинг 10.17. Содержимое файла auction/client/karma-test-runner.js

```
Error.stackTraceLimit = Infinity;
require('reflect-metadata/Reflect.js');
require('zone.js/dist/zone.js');
require('zone.js/dist/long-stack-trace-zone.js');
require('zone.js/dist/proxy.js');
require('zone.js/dist/sync-test.js');
```

```
require('zone.js/dist/jasmine-patch.js');
require('zone.js/dist/async-test.js');
require('zone.js/dist/fake-async-test.js');
var testing = require('@angular/core/testing');
var browser = require('@angular/platform-browser-dynamic/testing');
testing.TestBed.initTestEnvironment(
  browser.BrowserDynamicTestingModule,
  browser.platformBrowserDynamicTesting());
Object.assign(global, testing);
var testContext = require.context('./src', true, /\.spec\.ts/);
function requireAll(requireContext) {
  return requireContext.keys().map(requireContext);
}
var modules = requireAll(testContext);
```

Далее показан код, содержащийся в файле `webpack.test.config.js` (листинг 10.18). Он упрощен для проведения тестирования, поскольку в ходе проверки создавать пакеты не нужно. Имеющийся в Webpack dev-сервер не нужен, так как в качестве сервера выступает само средство Karma.

Листинг 10.18. Содержимое файла auction/client/webpack.test.config.js

```
const path         = require('path');
const DefinePlugin = require('webpack/lib/DefinePlugin');
const ENV  = process.env.NODE_ENV = 'development';
const HOST = process.env.HOST || 'localhost';
const PORT = process.env.PORT || 8080;
const metadata = {
  env : ENV,
  host: HOST,
  port: PORT
};
module.exports = {
  debug: true,
  devtool: 'source-map',
  module: {
    loaders: [
      {test: /\.css$/,  loader: 'raw', exclude: /node_modules/},
      {test: /\.css$/,  loader: 'style!css?-minimize', exclude: /src/},
      {test: /\.html$/, loader: 'raw'},
      {test: /\.ts$/,   loaders: [
        {loader: 'ts', query: {compilerOptions: {noEmit: false}}},
        {loader: 'angular2-template'}
      ]}
    ]
  },
  plugins: [
    new DefinePlugin({'webpack': {'ENV': JSON.stringify(metadata.env)}})
  ],
  resolve: { extensions: ['.ts', '.js']}
  }
};
```

В файле `client/package.json` находится следующее содержимое, имеющее отношение к Karma (листинг 10.19).

Листинг 10.19. Содержимое файла auction/client/package.json

```
"scripts": {
...
"test": "karma start karma.conf.js"
}
...
"devDependencies": {
...
    "karma": "^1.2.0",
    "karma-chrome-launcher": "^2.0.0",
    "karma-firefox-launcher": "^1.0.0",
    "karma-jasmine": "^1.0.2",
    "karma-mocha-reporter": "^2.1.0",
    "karma-webpack": "^1.8.0",
}
```

Для запуска тестов вручную введите команду `npm test` в окне команд, находясь при этом в каталоге `client`. Вы увидите вывод, похожий на тот, что показан на рис. 10.13.

```
START:
ts-loader: Using typescript@2.0.2 and /Users/yfain11/Documents/core-angular2/cod
e/chapter10/auction/client/tsconfig.json
02 09 2016 14:55:30.472:INFO [karma]: Karma v1.2.0 server started at http://loca
lhost:9876/
02 09 2016 14:55:30.474:INFO [launcher]: Launching browsers Chrome, Firefox with
 unlimited concurrency
02 09 2016 14:55:30.480:INFO [launcher]: Starting browser Chrome
02 09 2016 14:55:30.494:INFO [launcher]: Starting browser Firefox
02 09 2016 14:55:31.925:INFO [Chrome 52.0.2743 (Mac OS X 10.11.5)]: Connected on
 socket /#mK-QDpNhQ2ySM8DpAAAA with id 79472394
02 09 2016 14:55:34.325:INFO [Firefox 48.0.0 (Mac OS X 10.11.0)]: Connected on s
ocket /#zIgegGsuW02MlVyiAAAB with id 7680185
  ApplicationComponent
    ✔ is successfully instantiated
  StarsComponent
    ✔ is successfully injected
    ✔ readonly property is true by default
    ✔ all stars are empty
    ✔ all stars are filled
    ✔ emits rating change event when readonly is false
  ProductService
    ✔ getProductById() should return Product with ID=1

Finished in 0.844 secs / 0.821 secs
```

Рис. 10.13. Запуск Karma

Чтобы интегрировать запуск Karma в процесс сборки, можно изменить команду `npm prebuild`, придав ей следующий вид:

```
"prebuild": "npm run clean && npm run test"
"build": "webpack ...",
```

Теперь запуск команды `npm run build` повлечет старт команды `prebuild`, которая приведет к очистке каталога `output`, запустит тесты, а затем выполнит сборку. Если какой-либо из тестов окажется не пройден, то команда сборки `build` запущена не будет.

Итак, последний практикум, предложенный в данной книге, завершился. Если бы это было настоящее приложение, то вы продолжили бы работу по тонкой настройке конфигурации сборки, применяя избирательный подход к тем файлам, которые хотелось бы включить в сборку или исключить из нее. Webpack является весьма сложным инструментом, предлагающим нескончаемые возможности для оптимизации пакетов. Команда разработчиков упорно работает над оптимизацией Angular-кода, используя компиляцию перед выполнением, и нас не удивит, если фреймворк добавит к коду вашего приложения всего лишь 50 Кбайт.

ПРИМЕЧАНИЕ

В исходный код, сопровождающий эту главу, включен каталог под названием extras, содержащий еще одну реализацию аукциона, в которой часть, относящаяся к Angular, была создана с применением Angular CLI. Загляните в раздел scripts файла angular-cli.json, чтобы увидеть, как добавить библиотеки сторонних разработчиков к проекту Angular CLI.

10.6. Резюме

В данной главе речь шла не о создании кода. Целью ставилось изучение вопросов оптимизации и упаковки кода для его развертывания. В сообществе приверженцев JavaScript имеется несколько весьма популярных утилит для автоматизации сборки и упаковки веб-приложений, но наш выбор пал на сочетание Webpack и npm-сценариев.

Webpack является весьма технологичным и сложным инструментом, но мы представили небольшие комбинации загрузчиков и дополнительных модулей, выполнявших наши задачи. Если вы ищете более сложную конфигурацию Webpack, то попробуйте angular2-webpack-starter (https://github.com/AngularClass/angular-starter).

Вот основные выводы этой главы.

- Процесс подготовки пакетов для развертывания и запуска сборки должен быть автоматизирован на ранних стадиях разработки.
- Чтобы свести к минимуму количество запросов, выполняемых браузером для загрузки вашего приложения, объединяйте код в небольшое количество пакетов для развертывания.

- Избегайте упаковки одного и того же кода более чем в один пакет. Храните программные среды сторонних разработчиков в отдельном пакете, чтобы другие пакеты вашего приложения могли им совместно пользоваться.
- Всегда создавайте отображения на исходный код, поскольку они позволят вести отладку исходного кода в TypeScript. Такие отображения не увеличивают размер кода приложения и создаются, только если в браузере открыта панель Developer Tools (Инструменты разработчика). Поэтому использование отображений на исходный код приветствуется даже в коммерческих сборках.
- Чтобы запустить задачи по сборке и развертыванию, воспользуйтесь npm-сценариями, поскольку создавать их довольно просто, а диспетчер пакетов npm уже установлен. Если ведется работа по подготовке сборки для более крупного и сложного проекта и вы чувствуете потребность в языке сценариев для описания различных сценариев сборки, то введите в обиход своего проекта такое средство, как Gulp.
- Для быстрого старта в разработке в среде Angular и TypeScript создайте свой первый проект, используя Angular CLI.

Приложение А. Общее представление о ECMAScript 6

ECMAScript — стандарт для языков написания сценариев на стороне клиента. Первый выпуск спецификации ECMAScript состоялся в 1997 г., а шестой был завершен в 2015 г. Стандарт ECMAScript реализован в нескольких языках, а самой популярной является его реализация в JavaScript. В этом приложении будет рассмотрена JavaScript-реализация ECMAScript 6 (ES6), также известная как ECMAScript 2015.

На момент написания этих строк полная поддержка спецификации ES6 обеспечивалась не всеми браузерами. Чтобы прояснить текущую ситуацию с ее поддержкой, можно зайти на сайт, предоставляющий сведения о совместимости с ECMAScript: http://kangax.github.io/compat-table/es6/. Но откладывать в долгий ящик разработку в соответствии со спецификацией ES6 вам не придется, поскольку для превращения кода ES6 в код версии ES5, поддерживаемый всеми браузерами, уже сегодня можно воспользоваться таким транспилятором, как Traceur (https://github.com/google/traceur-compiler) или Babel (https://babeljs.io).

ПРИМЕЧАНИЕ

> Чтобы протестировать свой код ES6 в грядущих версиях популярных браузеров, загрузите самую последнюю тестовую сборку (nightly build) браузера Firefox с сайта https://www.mozilla.org/ru/firefox/channel/desktop/#nightly или воспользуйтесь удаленной версией Internet Explorer, находящейся на https://remote.modern.ie. Можно также взять с http://dev.chromium.org/getting-involved/dev-channel так называемую канареечную сборку (Canary build) браузера Chrome. При этом может понадобиться включить экспериментальные версии функциональных свойств JavaScript, прочитав об этом по адресу chrome://flags (для Chrome) или about://flags (для IE).

Предполагая, что вы уже знакомы с синтаксисом и API ES5, мы избирательно рассмотрим только новые функциональные возможности, появившиеся в ES6. Если вы в программировании на JavaScript немного отстали от новых веяний, то прочитайте интернет-версию приложения книги *Enterprise Web Development* Якова Файна (Yakov Fain) и др. (O'Reilly, 2014), доступную на GitHub: https://github.com/oreillymedia/enterprise_web_development. В 2016 г. была выпущена спецификация для ES7 (или в ином варианте названия ES 2016). Она невелика по объему. Спецификация языка ECMAScript 2017 опубликована на https://tc39.github.io/ecma262.

В данном приложении часто будут показываться фрагменты кода на ES5 и их эквиваленты на ES6. Но в ES6 не существует запретов на какой-либо устаревший синтаксис, поэтому в будущих версиях браузеров или в средах автономных JavaScript-интерпретаторов можно будет совершенно свободно запускать устаревший код на ES5 или ES3.

А.1. Порядок запуска примеров кода

Примеры кода для данного приложения даются в виде простых HTML-файлов, включающих сценарии на ES6, поэтому их можно запускать и отлаживать в инструментарии разработчика вашего браузера при условии, что он полностью поддерживает ES6. При отсутствии такой поддержки у вас есть несколько вариантов.

- Воспользоваться сайтом ES6 Fiddle, позволяющим копировать и вставлять фрагменты кода ES6 в поле, расположенное в левой части окна, нажимать кнопку Play (Запуск) и просматривать информацию, выводимую в консоли в правой части окна (рис. А.1).

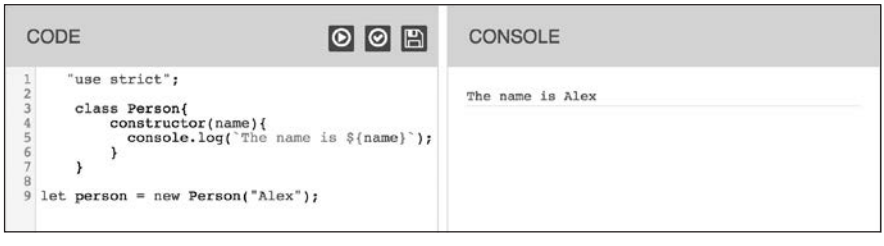


Рис. А.1. Применение ES6 Fiddle

- Задействовать транспилятор Traceur или Babel, чтобы преобразовать свой код из ES6 в ES5. Интерактивное средство, позволяющее быстро запускать фрагменты кода, называется Read-Eval-Print-Loop (REPLs). Им можно воспользоваться из Traceur (http://google.github.io/traceur-compiler/demo/repl.html#) или из Babel (http://babeljs.io/repl). На рис. А.2 слева в окне можно увидеть код, написанный на ES6, а справа показан его созданный средой ES5-эквивалент. Информация, выводимая в консоли при выполнении кодового примера (если таковая имеется), показана в нижней правой части экрана.

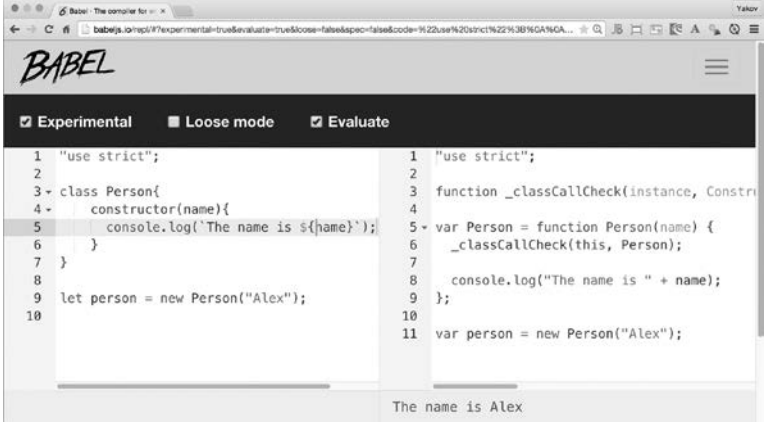


Рис. А.2. Использование Babel REPL

А.2. Литералы шаблонов

В ES6 введен новый синтаксис для работы со строковыми литералами, которые могут содержать встроенные выражения. Это свойство называется *строковой интерполяцией*.

В ES5 для создания строки, содержащей строковые литералы, перемешанные со значениями переменных, приходилось использовать объединение:

```
var customerName = "John Smith";
console.log("Hello" + customerName);
```

В ES6 литералы шаблонов заключены в символы обратных кавычек. Встраивать выражения в литералы можно путем заключения их в фигурные скобки, предваряемые знаком доллара. В следующем примере кода показана вставка значения переменной `customerName` в строковый литерал:

```
var customerName = "John Smith";
console.log(`Hello ${customerName}`);
function getCustomer(){
  return "Allan Lou";
}
console.log(`Hello ${getCustomer()}`);
```

Информация, выводимая эти кодом, будет иметь следующий вид:

```
Hello John Smith
Hello Allan Lou
```

В этом примере сначала в литерал шаблона было вставлено значение переменной `customerName`, а затем в него было включено значение, возвращенное функцией `getCustomer()`. Внутри фигурных скобок может быть встроено любое допустимое выражение JavaScript.

А.2.1. Строки с переносами

Строки могут располагаться на нескольких строчках вашего кода. Применяя обратные кавычки, можно записывать строки с переносами, не нуждаясь в их объединении или в использовании символа обратного слеша:

```
var message = `Please enter a password that
               has at least 8 characters and
               includes a capital letter`;
console.log(message);
```

В получающейся строке все пробелы будут рассматриваться в качестве части строки, поэтому на выходе будет показана следующая информация:

```
Please enter a password that
               has at least 8 characters and
               includes a capital letter
```

А.2.2. Тегированные шаблонные строки

Если строке шаблона предшествует имя функции, то сначала строка вычисляется, а затем передается функции для дальнейшей обработки. Строковые части шаблона задаются как массив, и все выражения, вычисляемые в шаблоне, передаются в виде отдельных аргументов. Синтаксис имеет несколько необычный вид, поскольку в нем не используются круглые скобки, как это делается при обычных вызовах функций:

```
mytag`Hello ${name}`;
```

Посмотрим, как этот синтаксис работает при выводе на стандартное устройство суммы со знаком валюты, который зависит от значения переменной `region`. Если значение переменной `region` равно `1`, то сумма не изменяется и перед ней ставится знак доллара. Если значением переменной `region` является `2`, то сумму следует преобразовать, применяя в качестве коэффициента обменного курса значение `0.9` и предваряя сумму знаком евро. Строка шаблона выглядит следующим образом:

```
`You've earned ${region} ${amount}!`
```

Вызовем тег-функцию `currencyAdjustment`. Тегированная строка шаблона имеет следующий вид:

```
currencyAdjustment`You've earned ${region} ${amount}!`
```

Функция `currencyAdjustment` получает три аргумента: первый из них представляет все строковые части из строки шаблона, второй представляет регион, а третий предназначается для суммы. После первого аргумента можно добавлять любое количество аргументов. Полноценный пример показан в листинге А.1.

Листинг А.1. Вывод суммы в валюте

```
function currencyAdjustment(stringParts, region, amount) {
    console.log( stringParts);
    console.log( region );
    console.log( amount );

  var sign;
  if (region==1){
    sign="$"
  } else{
    sign='\u20AC';  <-- Знак евро
    amount=0.9*amount;  <-- Конвертация в евро с 0,9 в качестве коэффициента обменного курса
  }
  return `${stringParts[0]}${sign}${amount}${stringParts[2]}`;
}

var amount = 100;
var region = 2;  <-- Европа: 2, США: 1.

var message = currencyAdjustment`You've earned ${region} ${amount}!`
console.log(message);
```

Функция `currencyAdjustment` получает строку со вставленными регионом и суммой и производит синтаксический разбор шаблона, отделяя строковые части от этих значений (пробелы также рассматриваются в качестве строковых частей). Для иллюстрации сначала выведем данные значения. Затем `currencyAdjustment` проверит регион, применит конвертацию и возвратит новый строковый шаблон. При запуске кода листинга А.1 на выходе будет получена следующая информация:

```
["You've earned "," ","!"]
2
100
You've earned €90!
```

Более подробные сведения о тегированных шаблонах можно получить, прочитав главу Template Literals в публикации *Exploring ES6*, принадлежащей Акселю Раушмайеру (Axel Rauschmayer), доступной на http://exploringjs.com.

А.3. Необязательные параметры и значения по умолчанию

В ES6 в качестве параметров (аргументов) функции можно указывать значения по умолчанию, которые станут использоваться, если соответствующее значение не будет предоставлено в ходе вызова функции. Предположим, что создается функция для вычисления налога, получающая два аргумента: годовой доход и штат в составе США, в котором проживает налогоплательщик. Если значение штата `state` не предоставлено, то нужно использовать значение `Florida`.

В ES5 тело функции пришлось бы начинать с проверки факта предоставления штата, и, если он не предоставлен, использовать `Florida`:

```
function calcTaxES5(income, state){
    state = state || "Florida";
    console.log("ES5. Calculating tax for the resident of " + state +
                                      " with the income " + income);
}
calcTaxES5(50000);
```

Этот код выведет следующую информацию:

```
"ES5. Calculating tax for the resident of Florida with the income 50000"
```

В ES6 можно указать значение по умолчанию непосредственно в сигнатуре функции:

```
function calcTaxES6(income, state = "Florida") {
  console.log("ES6. Calculating tax for the resident of " + state +
                                    " with the income " + income);
}
calcTaxES6(50000);
```

На выходе получится такая же информация:

```
"ES6. Calculating tax for the resident of Florida with the income 50000"
```

Вместо предоставления для необязательного параметра жестко заданного значения можно даже вызвать функцию, возвращающую нужное значение:

```
function calcTaxES6(income, state = getDefaultState()) {
  console.log("ES6. Calculating tax for the resident of " + state +
  ➥ " with the income " + income);
}
function getDefaultState(){
  return "Florida";
}
```

Однако нужно иметь в виду: функция `getDefaultState()` будет вызываться при каждом вызове функции `calcTaxES6()`, что может негативно отразиться на производительности. Этот новый синтаксис для необязательных параметров позволяет воспользоваться более лаконичным и понятным кодом.

A.4. Область видимости переменных

В ES5 используется весьма запутанный механизм области видимости. Независимо от места объявления переменной с помощью ключевого слова `var`, это объявление перемещается на вершину области видимости. Данный эффект называется *поднятием* (hoisting). А применение ключевого слова `this` не всегда толкуется так же однозначно, как в языках Java или C#.

В ES6 за счет введения ключевого слова `let` эта путаница с поднятием устранена (о чем пойдет речь в следующем подразделе), а от неразберихи, связанной с ключевым словом `this`, позволяют избавиться функции, обозначаемые в виде стрелок. Рассмотрим проблемы поднятия и использования ключевого `this` более подробно.

A.4.1. Поднятие переменных

В JavaScript все объявления переменных «всплывают» наверх, при этом блочная область видимости отсутствует. Рассмотрим следующий простой пример: внутри цикла `for` объявляется переменная `i`, которая тем не менее используется и за пределами данного цикла.

```
function foo(){
  for(var i=0;i<10;i++){
  }
  console.log("i=" + i);
}
foo();
```

При запуске этого кода будет выведено `i=10`. Переменная `i` все еще доступна за пределами цикла, несмотря на полное впечатление о том, что она предназначена исключительно для использования внутри него. JavaScript автоматически поднимает объявление переменной наверх.

В данном примере такое поднятие не приносит вреда, поскольку в нем имеется только одна переменная с именем `i`. Но если две переменные с одним и тем же именем объявлены внутри и за пределами какой-либо функции, то это может привести к запутанному поведению сценария. Рассмотрим код листинга А.2, где переменная `customer` объявляется в глобальной области видимости. А чуть ниже в локальной области видимости вводится еще одна переменная `customer`, но пока она закомментирована.

Листинг А.2. Поднятие объявления переменной

```
<!DOCTYPE html>
<html>
<head>
    <title>hoisting.html</title>
</head>
<body>
<script>
    "use strict";
    var customer = "Joe";
    (function (){
        console.log("The name of the customer inside the function is " +
        ➥ customer);
       /*   if (2 > 1) {
              var customer = "Mary";
         }*/
    })();
    console.log("The name of the customer outside the function is " +
    ➥ customer);
</script>
</body>
</html>
```

Откройте этот файл в браузере Chrome и посмотрите на информацию, выводимую в консоли на панели Developer Tools (Инструменты разработчика). На рис. А.3 показано, что глобальная переменная `customer` видна как внутри, так и за пределами функции.

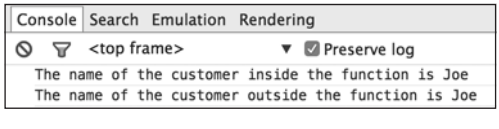


Рис. А.3. Объявление переменной поднято

Уберите знаки комментария, в которые заключена инструкция `if`, содержащая объявление и инициализацию переменной `customer` внутри фигурных скобок. Теперь у вас имеются две переменные с одним и тем же именем: одна в глобальной области видимости, а другая — в области видимости функции. Обновите страницу в браузере. Как показано на рис. А.4, информация, выводимая в консоли, изменится: переменная `customer`, находящаяся в области видимости функции, будет иметь значение `undefined`.

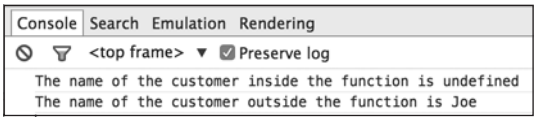


Рис. А.4. Инициализация переменной не поднята

Причиной такого поведения является то, что в ES5 объявления переменных поднимаются наверх области видимости, а инициализации переменных нет. Поэтому объявление второй инициализированной переменной `customer` было поднято на верх функции, и инструкция `console.log()` вывела значение, определенное внутри функции, которое затенило значение глобальной переменной `customer`.

Объявления функций также поднимаются, поэтому функцию можно вызвать до ее объявления:

```
doSomething();
function doSomething(){
    console.log("I'm doing something");
}
```

С другой стороны, функциональные выражения считаются инициализациями переменных, поэтому не поднимаются. Следующий фрагмент кода выдаст для переменной `doSomething` значение `undefined`:

```
doSomething();
var doSomething = function(){
    console.log("I'm doing something");
}
```

Теперь посмотрим, что в понятиях области видимости изменилось в ES6.

A.4.2. Блочная область видимости с использованием let и const

Объявление переменных с применением имеющегося в ES6 ключевого слова `let` вместо `var` позволяет переменным иметь блочную область видимости. Рассмотрим пример (листинг А.3).

Листинг А.3. Переменные с блочной областью видимости

```
<!DOCTYPE html>
<html>
<head>
    <title>let.html</title>
</head>
<body>
<script>
    "use strict";
    let customer = "Joe";
    (function (){
        console.log("The name of the customer inside the function is " +
        ➥ customer);
```

```
        if (2 > 1) {
          let customer = "Mary";
          console.log("The name of the customer inside the block is " +
          ➥ customer);
        }
    })();
    for (let i=0; i<5; i++){
        console.log("i=" + i);
    }
    console.log("i=" + i); //
      prints Uncaught ReferenceError: i is not defined
</script>
</body>
</html>
```

Теперь, как показано на рис. А.5, у двух переменных `customer` имеются разные области видимости и значения.

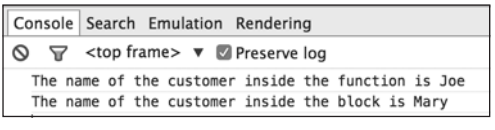


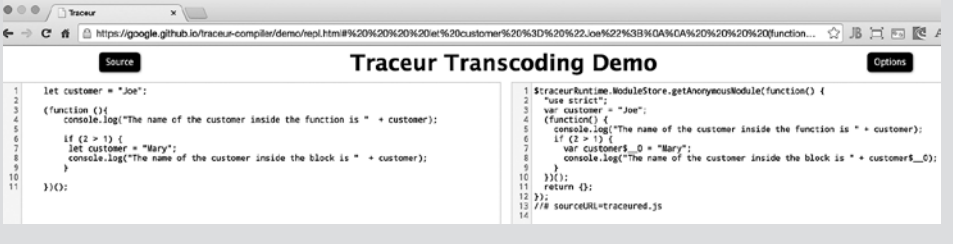
Рис. А.5. Блочная область видимости, получаемая с помощью let

Если переменная объявляется с применением `let` в цикле, то она будет доступна только внутри цикла:

```
for (let i=0; i<5; i++){
  console.log("i=" + i);
}
console.log("i=" + i); // ReferenceError: i is not defined
```

Тестирование ключевого слова let в Traceur REPL

Чтобы получить представление о внешнем виде транспилированного кода, зайдите на веб-страницу Traceur Transcoding Demo (http://google.github.io/traceur-compiler/demo/repl.html#), позволяющую вводить код в синтаксисе ES6 и преобразовывать (транскодировать) его в код в синтаксисе ES5 в интерактивном режиме. Перенесите код из листинга А.3 в левое текстовое поле, и, как показано на рисунке, в правом текстовом поле появится его ES5-версия.



Транспиляция ES6 в ES5 с использованием Traceur REPL

Как видите, средство Traceur ввело отдельную переменную `customer$_0`, чтобы отличить ее от переменной `customer`. Откройте при работе с Traceur REPL веб-консоль вашего браузера и тут же увидите результаты выполнения вашего кода.

Проще говоря, при разработке нового приложения вместо ключевого слова `var` используйте `let`. Оно позволит вам присваивать и переприсваивать значение переменной столько раз, сколько понадобится.

Если нужно объявить переменную, не способную изменять свое значение, то объявите ее с помощью ключевого слова `const`. Для констант также поддерживается блочная область видимости.

ПРИМЕЧАНИЕ

Единственное отличие let от const — последнее ключевое слово не позволяет присвоенному значению изменяться. Вам лучше всего использовать в своих программах const; если окажется, что эта переменная нуждается в изменении, то укажите вместо const ключевое слово let.

A.4.3. Блочная область видимости для функций

Если внутри блока (внутри фигурных скобок) объявить функцию, то за пределами блока она будет не видна. При выполнении следующего кода будет выдана ошибка с сообщением doSomething is not defined (doSomething не определена):

```
{
  function doSomething(){
    console.log("In doSomething");
  }
}
doSomething();
```

В ES5 объявление `doSomething()` будет поднято, и при выполнении кода появится сообщение In doSomething. Объявление функции внутри блока при использовании синтаксиса ES5 не рекомендовалось (см. публикацию ES5 Implementation Best Practice на http://mng.bz/Bvym), поскольку такое действие способно было привести к выдаче разных результатов на различных браузерах, которые могли проводить разбор этого синтаксиса по-разному.

A.5. Стрелочные функции, this и that

В ES6 введены выражения стрелочных функций, предоставляющих краткую форму записи для безымянных функций и добавляющих лексическую область видимости для переменной `this`. В некоторых других языках программирования (например, C# и Java) похожий синтаксис называется *лямбда-выражениями*.

Синтаксис стрелочной функции состоит из аргументов, знака жирной стрелки (`=>`), и тела функции. Если последнее состоит всего лишь из одного выражения, то вам

даже не понадобятся фигурные скобки. При условии, что функция, состоящая из одного выражения, возвращает значение, указывать инструкцию `return` не нужно — результат возвращается подразумеваемым образом:

```
let sum = (arg1, arg2) => arg1 + arg2;
```

Тело выражения стрелочной функции, составленное из нескольких строк, нужно заключать в фигурные скобки и использовать явное указание инструкции `return`:

```
(arg1, arg2) => {
  // выполнение какой-либо операции
  return someResult;
}
```

При отсутствии у стрелочной функции аргументов используются пустые круглые скобки:

```
() => {
  // выполнение какой-либо операции
  return someResult;
}
```

Если у функции всего лишь один аргумент, то круглые скобки не нужны:

```
arg1 => {
  // выполнение какой-либо операции
}
```

В следующем фрагменте кода выражение стрелочной функции передается в качестве аргумента методу `reduce()`, принадлежащему массиву, для вычисления суммы, а также методу `filter()` для вывода четных чисел:

```
var myArray = [1,2,3,4,5];
console.log( "The sum of myArray elements is " +
                 myArray.reduce((a,b) => a+b)); // выводит 15
console.log( "The even numbers in myArray are " +
                 myArray.filter( value => value % 2 == 0)); // выводит 2 4
```

Теперь, после ознакомления с синтаксисом стрелочных функций, посмотрим, как они оптимизируют работу с объектом `this`.

Определить в ES5, на какой из объектов ссылается ключевое слово `this`, порой становится весьма сложной задачей. Поищите в Интернете информацию по запросу `JavaScript this and that` и увидите множество статей, в которых люди жалуются на то, что `this` указывает на «не тот» объект. У ссылки `this` могут быть различные значения в зависимости от того, как вызвана функция и был ли использован строгий режим — strict mode (см. документацию раздела Strict Mode на https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode). Сначала мы опишем суть проблемы, а затем пути ее решения, предлагаемые ES6.

Рассмотрим код в файле `thisAndThat.html`, в котором каждую секунду вызывается функция `getQuote()`(листинг А.4). Она выводит случайно сгенерирован-

ную котировку для символа акции, предоставляемого функцией-конструктором `StockQuoteGenerator()`.

Листинг А.4. Содержимое файла thisAndThat.html

```
<!DOCTYPE html>
<html>
<head>
    <title>thisAndThat.html</title>
</head>
<body>
<script>
    function StockQuoteGenerator(symbol){
        // this.symbol = symbol;
        // внутри getQuote() этот объект не определен
        var that = this;
        that.symbol = symbol;
        setInterval(function getQuote(){
            console.log("The price quote for " + that.symbol
                    + " is " + Math.random());
        }, 1000);
    }
    var stockQuoteGenerator = new StockQuoteGenerator("IBM");
</script>
</body>
</html>
```

В закомментированной строке показан неправильный способ использования `this`, когда значение необходимо в функции, в которой, казалось бы, имеется такая же ссылка `this`, но это не так. Если значение переменной `this` не было сохранено в `that`, то значение `this.symbol` в функции `getQuote()`, вызванной в функции `setInterval()` в качестве функции обратного вызова, будет не определено. В функции `getQuote()` ключевое слово `this` указывает на глобальный объект, отличающийся от того объекта, на который указывает то же ключевое слово, определенное функцией-конструктором `StockQuoteGenerator()`.

Другим возможным решением, гарантирующим выполнение функции в конкретном объекте `this`, является применение функций JavaScript `call()`, `apply()` или `bind()`.

ПРИМЕЧАНИЕ

Если проблема, связанная с использованием ключевого слова this в языке программирования JavaScript, вам не знакома, то прочитайте статью Ричарда Бовелла (Richard Bovell) Understand JavaScript's 'this' with Clarity and Master It (http://javascriptissexy.com/understand-javascripts-this-with-clarity-and-master-it/).

В файле `fatArrow.html` (листинг А.5) показано решение с использованием стрелочной функции, исключающее необходимость сохранять `this` в `that`, как это делалось в коде файла `thisAndThat.html`.

Листинг А.5. Содержимое файла fatArrow.html

```
<!DOCTYPE html>
<html>
<head>
    <title>fatArrow.html</title>
</head>
<body>
<script>
  "use strict";
  function StockQuoteGenerator(symbol){
      this.symbol = symbol;
      setInterval(() => {
                  console.log("The price quote for " + this.symbol
                          + " is " + Math.random());
              }, 1000);
}
var stockQuoteGenerator = new StockQuoteGenerator("IBM");
</script>
</body>
</html>
```

Стрелочная функция, предоставляемая функции `setInterval()` в качестве аргумента, использует значение `this` окружающего контекста, поэтому распознает `IBM` в качестве значения выражения `this.symbol`.

А.5.1. Операторы rest и spread

В ES5 написание функции с переменным числом параметров требует использования специального объекта `arguments`. Этот объект *похож* на массив и содержит значения, соответствующие аргументам, передаваемым функции. Подразумеваемая переменная `arguments` может рассматриваться в любой функции в качестве локальной переменной.

В ES6 имеются операторы предоставления остальных аргументов `rest` и разложения массива на элементы `spread`, и оба они представлены многоточием (…). Оператор `rest` служит для передачи функции переменного числа аргументов и должен быть последним в списке аргументов. Если название аргумента функции начинается с многоточия, то все остальные аргументы будут получены в массиве.

Например, функции можно передать нескольких клиентов, используя только одно имя переменной с помощью оператора `rest`:

```
function processCustomers(...customers){
  // здесь находится реализация функции
}
```

В этой функции могут обрабатываться данные `customers` аналогично обработке любого массива. Предположим, что нужно создать функцию для вычисления налогов, вызываемую с первым аргументом, `income`, за которым следует любое количество аргументов, представляющих имена клиентов. В листинге А.6 показано, как можно обработать переменное количество аргументов с помощью сначала старого, а затем нового синтаксиса. В функции `calcTaxES5()` используется объект по имени `arguments`, а в функции `calcTaxES6()` — имеющийся в ES6 оператор `rest`.

Листинг A.6. Содержимое файла rest.html

```
<!DOCTYPE html>
<html>
<head>
    <title>rest.html</title>
</head>
<body>

<script>

  "use strict";

// ES5 и объект arguments
  function calcTaxES5(){

      console.log("ES5. Calculating tax for customers with the income ",
                          arguments[0]);

      var customers = [].slice.call(arguments, 1);

      customers.forEach(function (customer) {
          console.log("Processing ", customer);
      });
  }

  calcTaxES5(50000, "Smith", "Johnson", "McDonald");
  calcTaxES5(750000, "Olson", "Clinton");

// ES6 и оператор rest
  function calcTaxES6(income, ...customers) {
      console.log("ES6. Calculating tax for customers with the income ",
      ➥ income);
      customers.forEach(function (customer) {
          console.log("Processing ", customer);
      });
  }

  calcTaxES6(50000, "Smith", "Johnson", "McDonald");
  calcTaxES6(750000, "Olson", "Clinton");

</script>

</body>
</html>
```

Аргумент income является первым элементом

Извлечение массива, начинающегося со второго элемента

Обе функции, и `calcTaxES5()`, и `calcTaxES6()`, выдают одинаковые результаты:

```
ES5. Calculating tax for customers with the income 50000
Processing Smith
Processing Johnson
Processing McDonald
ES5. Calculating tax for customers with the income 750000
Processing Olson
```

```
Processing Clinton
ES6. Calculating tax for customers with the income 50000
Processing Smith
Processing Johnson
Processing McDonald
ES6. Calculating tax for customers with the income 750000
Processing Olson
Processing Clinton
```

Но обработка клиентов различается. Поскольку объект `arguments` не является настоящим массивом, в версии ES5 приходится создавать массив, задействуя методы `slice()` и `call()` для извлечения имен клиентов, начинающихся со второго элемента в `arguments`. В версии ES6 применять такие приемы не нужно, поскольку оператор `rest` предоставляет обычный массив клиентов. Использование остальных аргументов упрощает код и облегчает его чтение.

Если оператор `rest` может превратить переменное число параметров в массив, то оператор `spread` способен сделать обратное: превратить массив в список аргументов. Предположим, что нужно создать функцию, которая будет вычислять налог для трех клиентов с заданным доходом. На этот раз число аргументов фиксировано, но клиенты находятся в массиве. Чтобы превратить его в список отдельных аргументов, можно воспользоваться оператором `spread`, обозначаемым многоточием (...) (листинг А.7).

Листинг А.7. Содержимое файла spread.html

```
<!DOCTYPE html>
<html>
<head>
    <title>spread.html</title>
</head>
<body>
<script>

  "use strict";

  function calcTaxSpread( customer1, customer2, customer3, income) {
      console.log("ES6. Calculating tax for customers with the income ",
      ➥ income);

      console.log("Processing ", customer1, customer2, customer3);
  }

  var customers = ["Smith", "Johnson", "McDonald"];

  calcTaxSpread(...customers, 50000); ◄—— Оператор spread

</script>

</body>
</html>
```

В данном примере вместо извлечения значений из массива `customers` с последующим предоставлением этих значений в качестве аргументов функции используется массив с оператором `spread`, как будто функции говорят: «Тебе нужны три аргумента, но я даю тебе массив. Разбей его на элементы». Обратите внимание: в отличие от оператора `rest` оператор `spread` не обязан быть последним в списке аргументов.

A.5.2. Генераторы

Когда браузер выполняет обычную функцию JavaScript, он непрерывно запускает череду команд, принадлежащих функции, пока эти команды не закончатся. А выполнение *функции-генератора* может быть приостановлено и возобновлено множество раз. Функция-генератор может уступать управление вызывающему сценарию, запускаемому в том же самом потоке. Как только при выполнении кода в функции-генераторе встретится ключевое слово `yield`, данное выполнение приостанавливается, и вызывающий сценарий может возобновить его, вызвав в отношении генератора метод `next()`.

Чтобы превратить обычную функцию в генератор, нужно поставить между ключевым словом `function` и именем функции звездочку. Рассмотрим пример:

```
function* doSomething(){
  console.log("Started processing");
  yield;
  console.log("Resumed processing");
}
```

При вызове этой функции ее код не выполняется сразу же, а возвращается специальный объект `Generator`, который служит в качестве итератора. Следующая строка кода не приведет к выводу на экран какой-либо информации:

```
var iterator = doSomething();
```

Чтобы запустить выполнение тела функции, нужно вызвать в отношении генератора метод `next()`:

```
iterator.next();
```

После этой строки кода функция `doSomething()` выведет строку `Started processing` и приостановится, поскольку встретит оператор `yield`. Повторный вызов метода `next()` приведет к выводу строки `Resumed processing`.

Функции-генераторы используются, когда нужно создать функцию, выдающую поток данных. Предположим, нужна функция для извлечения и выдачи цен акций для указанного символа (IBM, MSFT и т. д.). Если цена акции падает ниже указанного значения (предельной цены), то вам нужно купить эту акцию.

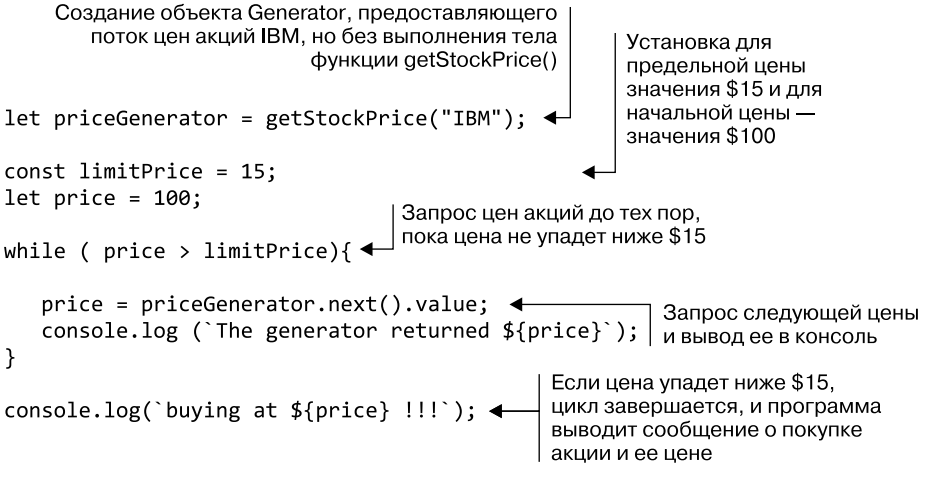
Данный сценарий имитирует следующая функция-генератор: `getStockPrice()` (листинг A.8). Чтобы ничего не усложнять, она не извлекает цены из фондовой биржи, создавая вместо этого случайные числовые значения с помощью метода `Math.random()`.

Листинг А.8. Содержимое функции getStockPrice()

```
function* getStockPrice(symbol){
  while(true){
    yield Math.random()*100;
    console.log(`resuming for ${symbol}`);
  }
}
```

Если после оператора `yield` имеется значение, то оно возвращается вызвавшему функцию коду, но на этом выполнение функции не завершается. Даже если в `getStockPrice()` имеется бесконечный цикл, то цена будет выдана (возвращена), только когда сценарий, вызвавший `getStockPrice()`, вызывает в отношении данного генератора метод `next()`, как в следующем коде (листинг А.9).

Листинг А.9. Вызов getStockPrice()



Запуск кода листинга А.9 выведет в консоли браузера некую информацию, похожую на представленную ниже:

```
The generator returned 61.63144460879266
resuming for IBM
The generator returned 96.60782956052572
resuming for IBM
The generator returned 31.163037824444473
resuming for IBM
The generator returned 18.416578718461096
resuming for IBM
The generator returned 55.80756475683302
resuming for IBM
The generator returned 14.203652134165168
buying at 14.203652134165168 !!!
```

Обратите внимание на порядок следования сообщений. Когда в отношении `priceGenerator` вызывается метод `next()`, выполнение приостановленного метода `getStockPrice()` возобновляется со строки кода, расположенной ниже оператора

`yield`, которая выводит `resuming for IBM`. Если даже поток управления покинет функцию, а затем вернется, то `getStockPrice()` будет помнить, что значением символа была строка `IBM`. Когда оператор `yield` возвращает управление за пределы сценария, он создает снимок стека; это позволяет ему запомнить все значения локальных переменных. Когда выполнение функции-генератора возобновится, данные значения не будут утрачены.

С помощью генераторов можно отделить реализацию конкретных операций (например, получение ценового предложения) от потребления данных, произведенных этими операциями. Потребитель данных в ленивом режиме вычисляет результаты и принимает решение, нужно ли запрашивать дополнительные данные.

A.5.3. Деструктурирование

Создание экземпляров объектов означает выстраивание их в памяти. *Деструктурирование* есть разделение объектов. В ES5 любой объект или коллекцию можно разобрать, создав занимающуюся этим функцию. В ES6 введен синтаксис деструктурирующего присваивания, позволяющий извлекать данные из свойств объекта или массива в простом выражении с указанием *шаблона соответствия*.

Деструктурирующее выражение состоит из шаблона соответствия, знака равенства и объекта или массива, подлежащего разделению. Проще всего все объяснить на примере, что мы и сделаем.

Деструктурирование объектов

Предположим, что функция `getStock()` возвращает объект `Stock`, имеющий атрибуты `symbol` и `price`. В ES5 при желании присвоить значения этих атрибутов отдельным переменным пришлось бы сначала вводить переменную для хранения объекта `Stock`, а затем создавать две инструкции присваивания атрибутов объекта соответствующим переменным:

```
var stock = getStock();
var symbol = stock.symbol;
var price = stock.price;
```

В ES6 нужно лишь создать шаблон соответствия в левой части выражения и присвоить ему объект `Stock`:

```
let {symbol, price} = getStock();
```

Видеть фигурные скобки слева от знака присваивания несколько необычно, но это часть синтаксиса выражения соответствия. Когда вы видите фигурные скобки слева, думайте о них как о блоке кода, а не как об объектном литерале.

В следующем сценарии (листинг A.10) показывается получение объекта `Stock` из функции `getStock()` и его деструктурирование в две переменные.

Листинг A.10. Деструктурирование объекта

```
function getStock(){
    return {
        symbol: "IBM",
```

```
        price: 100.00
    };
}
let {symbol, price} = getStock();
console.log(`The price of ${symbol} is ${price}`);
```

При запуске этого сценария будет выведена следующая информация:

```
The price of IBM is 100
```

Иными словами, в одном выражении присваивания получается привязка комплекта данных (в нашем случае атрибутов объекта) к набору переменных (`symbol` и `price`). Даже если у объекта `Stock` имеется больше двух атрибутов, это выражение деструктурирования все равно продолжит работать, поскольку `symbol` и `price` будут соответствовать шаблону. В выражении соответствия перечисляются только те переменные для атрибутов объекта, к которым проявляется интерес.

Код в листинге А.10 работает по причине совпадения имен переменных с именами атрибутов объекта `Stock`. Заменим имя `symbol` именем `sym`:

```
let {sym, price} = getStock();
```

Теперь выводимая информация изменится, поскольку JavaScript не знает, что значение принадлежащего объекту атрибута `symbol` должно быть присвоено переменной `sym`:

```
The price of undefined is 100
```

Это пример неверного шаблона соответствия. Если действительно нужно отобразить переменную по имени `sym` на атрибут `symbol`, то введите для имени `symbol` псевдоним:

```
let {symbol: sym, price} = getStock();
```

Если предоставить в левой части переменных больше, чем атрибутов у объекта, то лишние переменные получат значение `undefined`. При добавлении слева переменной `stockExchange` она будет проинициализирована значением `undefined`, поскольку в объекте, возвращенном методом `getStock()`, атрибут с таким именем отсутствует:

```
let {sym, price, stockExchange} = getStock();
console.log(`The price of ${symbol} is ${price} ${stockExchange}`);
```

В случае применения предыдущего деструктурирующего присваивания к тому же объекту `Stock` вывод на консоль будет иметь следующий вид:

```
The price of IBM is 100 undefined
```

Если нужно, чтобы у переменной `stockExchange` было значение по умолчанию, например, `NASDAQ`, то можно переписать деструктурирующее выражение следующим образом:

```
let {sym, price, stockExchange="NASDAQ"} = getStock();
```

Можно также выполнить деструктурирование вложенных объектов. В листинге А.11 создается вложенный объект, представляющий акцию Microsoft и пере-

даваемый функции `printStockInfo()`, которая извлекает из этого объекта символ акции и имя фондовой биржи.

Листинг A.11. Деструктурирование вложенного объекта

```
let msft = {symbol: "MSFT",
    lastPrice: 50.00,
    exchange: {
        name: "NASDAQ",
        tradingHours: "9:30am-4pm"
    }
};
function printStockInfo(stock){
    let {symbol, exchange:{name}} = stock;
    console.log(`The ${symbol} stock is traded at ${name}`);
}
printStockInfo(msft);
```

При запуске этого сценария будет выведена следующая информация:

```
The MSFT stock is traded at NASDAQ
```

Деструктурирование массивов

Работает во многом похоже на деструктурирование объектов, но вместо фигурных скобок используются квадратные. При деструктурировании объектов необходимо указывать переменные, соответствующие атрибутам, а при работе с массивами нужно определять переменные, соответствующие индексам. Следующий код извлекает значения двух элементов массива в две переменные:

```
let [name1, name2] = ["Smith", "Clinton"];
console.log(`name1 = ${name1}, name2 = ${name2}`);
```

Выводимая информация выглядит следующим образом:

```
name1 = Smith, name2 = Clinton
```

При необходимости извлечь только второй элемент этого массива шаблон соответствия имеет следующий вид:

```
let [, name2] = ["Smith", "Clinton"];
```

Если функция возвращает массив, то применение синтаксиса деструктурирования превращает ее в функцию с несколькими возвращаемыми значениями, как показано в функции `getCustomers()`:

```
function getCustomers(){
  return ["Smith", , , "Gonzales"];
}
let [firstCustomer,,,lastCustomer] = getCustomers();
console.log(`The first customer is ${firstCustomer} and the last one is
➥ ${lastCustomer}`);
```

А теперь объединим деструктурирование массива с остальными параметрами. Предположим, что имеется массив, состоящий из имен нескольких клиентов,

но нужно обработать только первые два. Как это делается, показано в следующем фрагменте кода:

```
let customers = ["Smith", "Clinton", "Lou", "Gonzales"];
let [firstCust, secondCust, ...otherCust] = customers;
console.log(`The first customer is ${firstCust} and the second one is
➥ ${secondCust}`);
console.log(`Other customers are ${otherCust}`);
```

Вывод в консоли, выполненный данным кодом, имеет следующий вид:

```
The first customer is Smith and the second one is Clinton
Other customers are Lou,Gonzales
```

Аналогичным образом, шаблон соответствия с параметром `rest` можно передать функции:

```
var customers = ["Smith", "Clinton", "Lou", "Gonzales"];
function processFirstTwoCustomers([firstCust, secondCust, ...otherCust]) {
  console.log(`The first customer is ${firstCust} and the second one is
  ➥ ${secondCust}`);
  console.log(`Other customers are ${otherCust}`);
}
processFirstTwoCustomers(customers);
```

Информация, выведенная в консоли, будет такой же:

```
The first customer is Smith and the second one is Clinton
Other customers are Lou,Gonzales
```

Подводя черту, можно отметить: преимущества деструктурирования заключаются в возможности писать меньше кода, когда необходимо инициализировать ряд переменных теми данными, которые находятся в свойствах объектов или в массивах.

A.6. Итерация с помощью forEach(), for-in и for-of

Циклический обход элементов коллекции объектов может быть выполнен с использованием различных ключевых слов и API JavaScript. В этом разделе будет показано применение нового цикла `for-of`. Мы сравним его с циклом `for-in` и методом `forEach()`.

A.6.1. Использование метода forEach()

Рассмотрим следующий код, выполняющий обход элементов массива, состоящего из четырех чисел. У этого массива также есть дополнительное свойство `description`, которое игнорируется методом `forEach()`:

```
var numbersArray = [1, 2, 3, 4];
numbersArray.description = "four numbers";
numbersArray.forEach((n) => console.log(n));
```

Информация, выводимая сценарием, имеет следующий вид:

```
1
2
3
4
```

Метод `forEach()` получает в качестве аргумента функцию и исправно выводит четыре числа из массива, игнорируя свойство `description`. Еще одно ограничение метода `forEach()` заключается в том, что он не позволяет прервать цикл преждевременно. Вместо `forEach()` приходится использовать метод `every()` или придумывать какие-либо другие ухищрения. Посмотрим, как здесь может помочь цикл `for-in`.

A.6.2. Использование цикла for-in

Данный цикл позволяет совершить обход *имен свойств* объектов и коллекций данных. В JavaScript любой объект является коллекцией пар «ключ — значение», где ключ представлен именем свойства, а значение — значением свойства. У массива имеется пять свойств: четыре для чисел и `description`. Выполним обход свойств этого массива:

```
var numbersArray = [1, 2, 3, 4];
numbersArray.description = "four numbers";
for (let n in numbersArray) {
  console.log(n);
}
```

Предыдущий код выведет следующую информацию:

```
0
1
2
3
description
```

Запуск данного кода в отладчике показывает: каждое из этих свойств является строкой. Чтобы посмотреть фактические значения свойств, вывод элементов массива нужно выполнить с помощью записи `numbersArray[n]`:

```
var numbersArray = [1, 2, 3, 4];
numbersArray.description = "four numbers";
for (let n in numbersArray) {
  console.log(numbersArray[n]);
}
```

Теперь вывод выглядит следующим образом:

```
1
2
3
4
four numbers
```

Как видите, цикл `for-in` позволяет обойти все свойства, а не только данные, в которых может быть не то, что вам нужно. А теперь применим новый синтаксис `for-of`.

А.6.3. Использование for-of

В ES6 введен новый цикл `for-of`, позволяющий проводить итерацию данных независимо от того, какие еще свойства имеются в коллекции данных. При необходимости этот цикл можно прервать, задействуя ключевое слово `break`:

```
var numbersArray = [1, 2, 3, 4];
numbersArray.description = "four numbers";
console.log("Running for of for the entire array");
for (let n of numbersArray) {
  console.log(n);
}
console.log("Running for of with a break");
for (let n of numbersArray) {
  if (n >2) break;
  console.log(n);
}
```

Этот сценарий выведет следующую информацию:

```
Running for of for the entire array
1
2
3
4
Running for of with a break
1
2
```

Цикл `for-of` работает с любым итерируемым объектом, включая `Array`, `Map`, `Set` и др. Строки также являются итерируемыми объектами. Следующий код посимвольно выводит содержимое строки `John`:

```
for (let char of "John") {
  console.log(char);
}
```

А.7. Классы и наследование

Как в ES3, так и в ES5 поддерживается объектно-ориентированное программирование и наследование. Но с помощью классов ES6 писать и читать код существенно проще.

В ES5 объекты могут создаваться либо с самого начала, либо путем наследования из других объектов. Изначально все объекты JavaScript наследуются от объекта `Object`. Данное наследование реализуется благодаря специальному свойству

prototype, которое указывает на предка того или иного объекта. Это называется *прототипным наследованием*. Например, чтобы создать объект NJTax, являющийся наследником объекта Tax, можно написать следующий код:

```
functionTax() {
  // Здесь находится код объекта tax
}

functionNJTax() {
  // Здесь находится код объекта NewJerseytax
}

NJTax.prototype = new Tax(); ◄── Наследование NJTax из Tax

var njTax = new NJTax();
```

В ES6 введены ключевые слова class и extends, чтобы поставить синтаксис в один ряд с синтаксисом других объектно-ориентированных языков, таких как Java и C#. В ES6 аналог предыдущего кода выглядит следующим образом:

```
class Tax {
  // Здесь находится код класса tax}
class NJTax extends Tax {
  // Здесь находится код объекта New Jersey tax
}
var njTax = new NJTax();
```

Класс Tax — класс-предок, или *суперкласс*, а NJTax — потомок, или *подкласс*. Можно также сказать, что класс NJTax имеет с классом Tax отношения типа is a («является»). Иными словами, NJTax является классом Tax. В NJTax можно реализовать дополнительные функциональные возможности, но NJTax по-прежнему «является» (is a) классом Tax или относится к его разновидности (is a kind of). Аналогично этому, если создать класс Employee, являющийся наследником класса Person, то можно сказать, что Employee — это Person.

Можно создать один или несколько экземпляров объектов:

```
                          ┌── Первый экземпляр объекта Tax
var tax1 = new Tax(); ◄──┘
var tax2 = new Tax(); ◄── Второй экземпляр объекта Tax
```

ПРИМЕЧАНИЕ

Объявления классов поднятию не подвергаются. Сначала нужно объявить класс и только потом работать с ним.

У каждого из этих объектов будут свойства и методы, имеющиеся в классе Tax, но у них будет другое *состояние*. Например, первый экземпляр может быть создан для клиента с годовым доходом $50 000, а второй — для клиента, заработавшего за год $75 000. Каждый экземпляр будет совместно использовать одни и те же копии методов, объявленных в классе Tax, исключая тем самым дублирование кода.

В ES5 можно также избежать дублирования кода, объявляя методы не внутри объектов, а в их прототипах:

```
function Tax() {
  // Здесь находится код объекта tax
}
Tax.prototype = {
  calcTax: function() {
    // Здесь находится код для вычисления налога (tax)
  }
}
```

JavaScript остается языком с прототипным наследованием, но ES6 позволяет создавать более элегантный код:

```
class Tax(){

  calcTax(){
    // Здесь находится код для вычисления налога (tax)
  }
}
```

Поддержка переменных элементов класса отсутствует

Синтаксис ES6 не позволяет объявлять переменные элементов класса, как в Java, C# или TypeScript. Следующий синтаксис *не* поддерживается:

```
class Tax {
  var income;
}
```

A.7.1. Конструкторы

В ходе создания экземпляров в классах выполняется код, помещенный в специальные методы под названием *конструкторы*. В таких языках, как Java и C#, у конструктора должно быть имя, совпадающее с именем класса, но в ES6 конструктор класса указывается с помощью ключевого слова `constructor`:

```
class Tax{
  constructor (income){
    this.income = income;
  }
}
var myTax = new Tax(50000);
```

Метод `constructor` имеет специализированный характер и выполняется только один раз при создании объекта. Если вам знаком синтаксис Java или C#, то предыдущий код может показаться несколько странным: в нем нет объявления отдельной переменной `income` на уровне класса, она создается в динамическом режиме в отноше-

нии объекта this, при этом this.income инициализируется значениями аргумента конструктора. Переменная this указывает на экземпляр текущего объекта.

В следующем примере показывается порядок создания экземпляра подкласса NJTax, где его конструктору предоставляется значение дохода, равное 50 000:

```
class Tax{
    constructor(income){
       this.income = income;
    }
}
class NJTax extends Tax{
    // Здесь находится код объекта New Jersey tax
}
var njTax = new NJTax(50000);
console.log(`The income in njTax instance is ${njTax.income}`);
```

Вывод этого фрагмента кода выглядит следующим образом:

```
The income in njTax instance is 50000
```

Поскольку в подклассе NJTax собственный конструктор не определяется, при создании экземпляра NJTax происходит автоматический вызов конструктора из родительского класса Tax. Если в подклассе определен собственный конструктор, то этого не произойдет. Соответствующий пример будет показан в подразделе А.7.4.

Обратите внимание: у вас есть возможность доступа к значению income за пределами класса через ссылочную переменную njTax. А можно ли скрыть income за пределами объекта? Этот вопрос мы рассмотрим в разделе А.9.

А.7.2. Статические переменные

Если вам нужно свойство класса, совместно используемое множеством экземпляров объекта, то необходимо создать его за пределами объявления класса. В следующем примере переменная counter совместно применяется обоими экземплярами объекта A:

```
class A{
  }
A.counter = 0;
var a1 = new A();
A.counter++;
console.log(A.counter);
var a2 = new A();
A.counter++;
console.log(A.counter);
```

Этот код выводит следующий результат:

```
1
2
```

А.7.3. Геттеры, сеттеры и определение методов

Синтаксис для объектов геттеров и сеттеров не является новшеством ES6, но изучим его, прежде чем перейти к новому синтаксису определения методов. Сеттеры и геттеры привязывают функции к свойствам объекта. Рассмотрим объявление и использование литерала объекта `Tax`:

```
var Tax = {
  taxableIncome:0,
  get income() {return this.taxableIncome;},
  set income(value){ this.taxableIncome=value}
};
Tax.income=50000;
console.log("Income: " + Tax.income); // выводит Income: 50000
```

Обратите внимание: присваивание и извлечение значения `income` выполняется с применением системы записи, использующей точку, точно так же, как при объявлении свойства объекта `Tax`.

В ES5 приходится применять ключевое слово `function`, например, `calculateTax = function(){…}`. ES6 позволяет в любом определении метода опустить ключевое слово `function`:

```
var Tax = {
    taxableIncome:0,
    get income() {return this.taxableIncome;},
    set income(value){ this.taxableIncome=value},
    calculateTax(){ return this.taxableIncome*0.13}
};
Tax.income=50000;

console.log(`For the income ${Tax.income} your tax is ${Tax.calculateTax()}`);
```

Ниже показана информация, выводимая этим кодом:

```
For the income 50000 your tax is 6500
```

Геттеры и сеттеры предлагают удобный синтаксис для работы со свойствами. Например, если будет решено добавить к геттеру `income` какой-нибудь проверочный код, то изменять сценарии, использующие форму записи `Tax.income`, не придется. Плохо только то, что в ES6 в классах не поддерживаются закрытые переменные, поэтому ничто не препятствует программисту получить доступ к переменной, применяемой в геттере или сеттере (например, к `taxableIncome`) напрямую. О сокрытии (инкапсуляции) переменных разговор пойдет в разделе А.9.

А.7.4. Ключевое слово super и функция super

Функция `super()` позволяет подклассу (потомку) вызывать конструктор из родительского класса (предка). Ключевое слово `super` служит для вызова метода, определенного в родительском классе. Использование функции `super()` и ключевого слова `super` проиллюстрировано в листинге А.12. У класса `Tax` име-

ется метод `calculateFederalTax()`, а в его подклассе `NJTax` добавляется метод `calculateStateTax()`. У обоих этих классов имеются свои собственные версии метода `calcMinTax()`.

Листинг А.12. Использование функции super() и ключевого слова super

```
"use strict";
class Tax{
    constructor(income){
        this.income = income;
    }
    calculateFederalTax(){
        console.log(`Calculating federal tax for income ${this.income}`);
    }
    calcMinTax(){
        console.log("In Tax. Calculating min tax");
        return 123;
    }
}
class NJTax extends Tax{
   constructor(income, stateTaxPercent){
        super(income);
        this.stateTaxPercent=stateTaxPercent;
    }
    calculateStateTax(){
        console.log(`Calculating state tax for income ${this.income}`);
    }
    calcMinTax(){
        super.calcMinTax();
        console.log("In NJTax. Adjusting min tax");
    }
}
var theTax = new NJTax(50000, 6);
theTax.calculateFederalTax();
theTax.calculateStateTax();
theTax.calcMinTax();
```

Запуск этого кода приведет к выводу следующей информации:

```
Calculating federal tax for income 50000
Calculating state tax for income 50000
In Tax. Calculating min tax
In NJTax. Adjusting min tax
```

У класса `NJTax` имеется свой, определенный явным образом конструктор с двумя аргументами, `income` и `stateTaxPercent`, предоставляемыми при создании экземпляра `NJTax`. Чтобы гарантировать вызов конструктора `Tax` (который устанавливает в объекте атрибут `income`), из конструктора подкласса явным образом вызывается `super("50000");`. Без данной строки кода фрагмент программы, показанный в листинге А.12, выдаст сообщение об ошибке и, даже если он этого не сделает, код в `Tax` не будет видеть значение `income`.

Если нужно вызвать конструктор родительского класса, то это нужно сделать в конструкторе подкласса путем вызова функции `super()`. Еще один способ вызова кода в родительском классе заключается в использовании ключевого слова `super`. Метод `calcMinTax()` имеется как в `Tax`, так и в `NJTax`. Метод, определенный в родительском классе `Tax`, вычисляет базовую минимальную сумму в соответствии с федеральными налоговыми законами, а версия данного метода из подкласса применяет базовое значение и проводит его уточнение. У обоих методов есть одинаковая сигнатура, поэтому имеет место *переопределение метода*.

С помощью вызова `super.calcMinTax()` гарантируется, что для вычисления налога штата берется в расчет базовый федеральный налог. Если не вызвать `super.calcMinTax()`, то на первый план выйдет переопределение метода и применена будет версия метода `calcMinTax()`, определенная в подклассе. Такое переопределение метода часто используется для замены его функциональных возможностей, определяемых в родительском классе, без изменения его кода.

Предостережение, касающееся классов и наследования

Классы в ES6 являются всего лишь синтаксической уловкой, улучшающей читаемость кода. «За кулисами» JavaScript по-прежнему использует прототипное наследование, позволяющее в динамическом режиме заменять предка в ходе выполнения сценария, а у класса может быть только один предок. Постарайтесь избегать создания глубоких иерархий наследования, поскольку они снижают гибкость вашего кода и усложняют его реструктуризацию в случае необходимости.

Хотя применение ключевого слова `super` или функции `super()` позволяет вызывать код предка, нужно избегать их использования, поскольку они приводят к жесткому связыванию между объектом-потомком и объектом-предком. Чем меньше потомок знает о своем предке, тем лучше. Если предок объекта изменится, то у его новой версии может не оказаться того метода, попытка вызова которого предпринимается с помощью функции `super()`.

А.8. Асинхронная обработка с помощью промисов

Для организации асинхронной обработки в предыдущих реализациях ECMAScript приходилось использовать *функции обратного вызова* (callbacks), представляющие собой функции, передаваемые в качестве аргументов другим функциям для вызова. Функции обратного вызова могут вызываться в синхронном или в асинхронном режиме.

В разделе А.6 функция обратного вызова передавалась функции `forEach()` для синхронного вызова. При создании Ajax-запросов к серверу функция об-

ратного вызова передается для асинхронного вызова, когда результаты поступают от сервера.

А.8.1. Ад обратного вызова

Рассмотрим пример получения данных с сервера о заказанных товарах. Все начинается с асинхронного обращения к серверу для получения информации о клиентах, а затем для каждого клиента нужно сделать еще один вызов, чтобы получить заказы. Для каждого из них нужно получить товары. При завершающем вызове будут получены подробные описания товаров.

При асинхронной обработке неизвестно, когда каждая из этих операций завершится, так что нужно создавать функции обратного вызова, которые вызываются после окончания работы их предшественников. Чтобы имитировать задержки, как будто на завершение каждой операции уходит одна секунда, воспользуемся функцией `setTimeout()` (листинг А.13).

Листинг А.13. Вложенные функции обратного вызова

```
function getProductDetails() {
    setTimeout(function () {
      console.log('Getting customers');
      setTimeout(function () {
         console.log('Getting orders');
         setTimeout(function () {
            console.log('Getting products');
            setTimeout(function () {
               console.log('Getting product details')
            }, 1000);
         }, 1000);
      }, 1000);
    }, 1000);
};
getProductDetails();
```

При запуске этого кода с односекундными задержками будут выведены следующие сообщения:

```
Getting customers
Getting orders
Getting products
Getting product details
```

Уровень вложенности, показанный в листинге А.13, уже затрудняет чтение кода. А теперь представим, что к этому добавляются логика приложения и обработка ошибок. Написание кода подобным образом зачастую называют *адом обратного вызова* или *гибельным треугольником* (пустые пространства в коде принимают форму треугольника).

А.8.2. Промисы ES6

В ES6 введены *промисы*, позволяющие избавиться от этой вложенности и улучшить читаемость кода, обеспечивая наличие тех же функциональных возможностей, что и у функций обратного вызова. Объект `Promise` ожидает и отслеживает результат асинхронной операции и позволяет узнать о ее успешном или аварийном завершении, чтобы у вас появилась возможность предпринять соответствующие последующие шаги. Объект `Promise` представляет будущий результат операции и может находиться в одном из следующих состояний:

- *выполнен* — операция успешно завершилась;
- *отклонен* — операция дала сбой и возвратила ошибку;
- *в ожидании* — операция находится в работе, не будучи ни выполненной, ни отклоненной.

Экземпляр объекта `Promise` создается путем предоставления его конструктору двух функций для вызова в двух случаях: при выполнении операции и при ее отклонении. Рассмотрим сценарий, использующий функцию `getCustomers()` (листинг А.14).

Листинг А.14. Использование промиса

```
function getCustomers(){

  return new Promise(
     function (resolve, reject){

      console.log("Getting customers");
        // Здесь имитируется вызов синхронного сервера
      setTimeout(function(){
        var success = true;
        if (success){
          resolve( "John Smith"); ◄——— Получаем клиента
        }else{
          reject("Can't get customers");
        }
      },1000);
    }
  );
}

let promise = getCustomers()
  .then((cust) => console.log(cust))
  .catch((err) => console.log(err));
console.log("Invoked getCustomers. Waiting for results");
```

Функция `getCustomers()` возвращает объект `Promise`, экземпляр которого создается с функцией, имеющей в качестве аргументов конструктора `resolve` и `reject`. В коде `resolve()` вызывается, если будет получена информация о клиенте. В целях упрощения функция `setTimeout()` имитирует асинхронный вызов, длящийся одну

секунду. Кроме того, жестко задается значение `true` для флага `success`. В настоящем сценарии можно выдать запрос с использованием объекта `XMLHttpRequest` и вызвать `resolve()` в случае успешного возвращения результата или `reject()` при возникновении ошибки.

В нижней части листинга А.14 к экземпляру `Promise()` прикрепляются методы `then()` и `catch()`. Из этих двух методов будет вызван только один. Вызов внутри функции `resolve("John Smith")` приводит к вызову метода `then()`, получающего в качестве аргумента `John Smith`. Если заменить значение `success` на `false`, то будет вызван метод `catch()` с аргументом `Can't get customers`.

Запуск кода листинга А.14 приведет к выводу в консоли следующих сообщений:

```
Getting customers
Invoked getCustomers. Waiting for results
John Smith
```

Обратите внимание: сообщение Invoked getCustomers. Waiting for results выводится перед John Smith. Тем самым подтверждается, что функция `getCustomers()` работает асинхронно.

Каждый промис соответствует одной асинхронной операции, и для обеспечения конкретного порядка выполнения промиса можно выстраивать в цепочку. Добавим функцию `getOrders()`, которая находит заказы, принадлежащие указанному клиенту, и составляет цепочку с функцией `getCustomers()` (листинг А.15).

Листинг А.15. Выстраивание промисов в цепочку

```
'use strict';

function getCustomers(){
  let promise = new Promise(
    function (resolve, reject){

     console.log("Getting customers");
       // Здесь имитируется вызов синхронного сервера
     setTimeout(function(){
let success = true;
        if (success){
          resolve( "John Smith"); ◄——— Получаем клиента
        }else{
          reject("Can't get customers");
        }
      },1000);

    }
  );
  return promise;
}

function getOrders(customer){

  let promise = new Promise(
```

```
    function (resolve, reject){

       // Здесь имитируется вызов синхронного сервера
         setTimeout(function(){
       let success = true;
       if (success){
         resolve( `Found the order 123 for ${customer}`);
       }else{
         reject("Can't get orders");
       }
     },1000);

    }
  );
  return promise;
}

getCustomers()
.then((cust) => {console.log(cust);return cust;})
  .then((cust) => getOrders(cust))
  .then((order) => console.log(order))
  .catch((err) => console.error(err));
console.log("Chained getCustomers and getOrders. Waitingforresults");
```

Получаем заказ

В этом коде не только объявляются и выстраиваются в цепочку две функции, но и демонстрируется способ вывода в консоли промежуточных результатов. Далее показан вывод, выполняемый кодом листинга А.15 (обратите внимание: клиент, возвращенный из функции `getCustomers()`, был должным образом передан функции `getOrders()`):

```
Getting customers
Chained getCustomers and getOrders. Waiting for results
John Smith
Found the order 123 for John Smith
```

Используя функцию `then()`, можно выстраивать в цепочку сразу несколько функций и при этом иметь только один сценарий обработки ошибок для всех цепочных вызовов. При возникновении ошибки она распространится по всей цепочке `then`, пока не найдет обработчик ошибки. После того как произойдет ошибка, никакие `then` уже вызываться не будут.

Если в листинге А.15 изменить значение переменной `success` на `false`, то выведется сообщение Can't get customers и метод `getOrders()` вызван не будет. При перемещении всех этих выводов в консоли код, извлекающий клиентов и заказы, станет выглядеть более чистым и понятным:

```
getCustomers()
  .then((cust) => getOrders(cust))
  .catch((err) => console.error(err));
```

Добавление `then` не снижает читаемость этого кода (сравните его с гибельным треугольником из листинга А.13).

А.8.3. Одновременное разрешение сразу нескольких промисов

Еще один рассматриваемый вопрос касается асинхронных функций, не зависящих друг от друга. Предположим, нужно вызвать две функции, не придерживаясь какого-либо определенного порядка, но при этом следует совершить некое действие только *после* того, как обе функции завершат работу. У объекта `Promise` имеется метод `all()`, получающий коллекцию промисов, которая допускает последовательный обход элементов и выполняет (разрешает) все эти промисы. Поскольку метод `all()` возвращает объект `Promise`, к результату можно добавить `then()` или `catch()` (или оба метода).

Посмотрим, что получится, если воспользоваться `all()` с функциями `getCustomers()` и `getOrders()`:

```
Promise.all([getCustomers(), getOrders()])
.then((order) => console.log(order));
```

При выполнении этого кода будет выведена следующая информация:

```
Getting customers
Getting orders for undefined
["John Smith","Order 123"]
```

Обратите внимание на сообщение Getting orders for undefined. Оно появилось из-за беспорядочности разрешения промисов, по причине которой функция `getOrders()` не получила клиента в качестве аргумента. Конечно, для данного сценария использование `Promise.all()` вряд ли подходит, но бывают и более подходящие для этого ситуации. Представим веб-портал, которому необходимо выполнить несколько асинхронных вызовов, чтобы получить сводку погоды, новости рынка акций и информацию об интенсивности движения транспорта. Если нужно, чтобы страница портала отображалась по завершении всех этих вызовов, то лучшего средства, чем `Promise.all()`, не найти:

```
Promise.all([getWeather(), getStockMarketNews(), getTraffic()])
.then(renderGUI);
```

По сравнению с функциями обратного вызова промисы могут сделать код более линейным и легче читаемым и соответствовать сразу нескольким состояниям приложения. К их недостаткам можно отнести невозможность их прекратить. Представим нетерпеливого пользователя, несколько раз нажимающего кнопку, чтобы получить данные от сервера. При каждом щелчке создается промис и инициируется HTTP-запрос. Способа сохранить только последний запрос и отменить незавершенные запросы не существует. Следующим этапом эволюции объекта `Promise` является объект `Observable`, который может появиться в будущих спецификациях ECMAScript; как он может использоваться уже сейчас, описано в главе 5.

ПРИМЕЧАНИЕ

Вскоре для получения ресурсов по сети вместо объекта XMLHttpRequest можно будет воспользоваться API Fetch. Он основан на применении промисов, а более подробные сведения о нем можно получить, изучив документацию Mozilla Developer Network (https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API).

А.9. Модули

В любом языке программирования разбиение кода на модули помогает выстроить приложение из логически независимых и по возможности многократно используемых модулей. Модульные приложения позволяют более эффективно распределять программируемые задачи между разработчиками программного кода. Разработчики должны решить, какие API должны встречаться в модулях для внешнего применения, а какие — для внутреннего.

В ES5 нет языковых конструкций для создания модулей, поэтому приходится прибегать к одному из следующих вариантов:

- реализовать шаблон конструкции модуля вручную в виде тут же инициализируемой функции (см. статью Тодда Мотто (Todd Motto) Mastering the Module Pattern на https://toddmotto.com/mastering-the-module-pattern/);
- воспользоваться сторонней реализацией стандарта AMD (https://en.wikipedia.org/wiki/Asynchronous_module_definition) или стандарта CommonJS (http://wiki.commonjs.org/wiki/CommonJS).

Стандарт CommonJS был создан для внедрения модульности в приложения JavaScript, запускаемые вне браузеров (например, тех, которые созданы в среде Node.js и разработаны под управлением Google-движка V8). AMD используется преимущественно для приложений, запускаемых в браузере.

В любом скромном по размеру веб-приложении следует минимизировать объем JavaScript-кода, загружаемого на стороне клиента. Представим обычный интернет-магазин. Нужно ли загружать код для обработки платежей, когда пользователь открывает главную страницу приложения? Что, если пользователи никогда не нажмут кнопку Place Order (Разместить заказ)? Было бы вполне разумно разбить приложение на модули для загрузки кода по мере необходимости. Require JS является, наверное, наиболее популярной библиотекой стороннего производителя, реализующей стандарт AMD. Она позволяет определять зависимости между модулями и загружать их в браузер по требованию.

Начиная с ES6, модули стали частью языка, что означает прекращение использования разработчиками сторонних библиотек для реализации различных стандартов. Даже если браузеры не поддерживают модули ES6 естественным образом, то существуют полифиллы, позволяющие приступить к использованию модулей JavaScript уже сейчас. В этой книге мы используем полифилл под названием SystemJS.

А.9.1. Импорт и экспорт данных

Как правило, модуль представляет собой обыкновенный файл с кодом JavaScript, в котором реализуется определенное функциональное свойство и предоставляется открытый интерфейс, позволяющий задействовать этот модуль другим программам на JavaScript. Специального ключевого слова для объявления кода конкретного файла в качестве модуля не существует. Но в сценарии можно воспользоваться ключевыми словами `import` и `export`, превращающими сценарий в модуль ES6.

Ключевое слово `import` позволяет одному сценарию объявлять свои потребности в применении переменных или функций, находящихся в другом сценарном файле. Аналогично этому ключевое слово `export` позволяет объявлять переменные, функции или классы, которые модуль может выставить для использования другими сценариями. Иными словами, с помощью ключевого слова `export` можно сделать выбранные API доступными другим модулям. Имеющиеся в модуле функции, переменные и классы, не объявленные явным образом экспортируемыми, остаются инкапсулированными в модуле.

ПРИМЕЧАНИЕ

Основным отличием модуля от обычного файла JavaScript является то, что при добавлении файла к странице с помощью тега <script> он становится частью глобального контекста, а объявления в модулях являются локальными и никогда не становятся частью глобального пространства имен. Даже экспортируемые элементы доступны только тем модулям, которые их импортируют.

В ES6 две разновидности применения ключевого слова `export`: именованное и по умолчанию. При именованном экспортировании можно воспользоваться ключевым словом `export`, поставив его перед несколькими элементами модуля (такими как классы, функции и переменные). Код в следующем файле (`tax.js`) экспортирует переменную `taxCode` и функцию `calcTaxes()`, но функция `doSomethingElse()` остается скрытой от внешних сценариев:

```
export var taxCode;
export function calcTaxes() { // сюда помещается код }
function doSomethingElse() { // сюда помещается код }
```

Когда сценарий импортирует поименованные элементы экспортируемого модуля, их имена должны быть заключены в фигурные скобки. Эта особенность показана в файле `main.js`:

```
import {taxCode, calcTaxes} from 'tax';
if (taxCode === 1) { // выполнение каких-либо операций }
calcTaxes();
```

Здесь `tax` ссылается на имя файла, содержащего модуль, за исключением расширения.

Один из экспортируемых элементов модуля может быть помечен как `default`, что означает безымянный экспорт, и другой модуль способен присвоить этому элементу в свой инструкции `import` любое имя.

Код файла `my_module.js`, экспортирующий функцию, может иметь следующий вид:

```
export default function() { // Выполнение каких-либо операций }

export var taxCode;
```

Нет точки с запятой

Код файла `main.js` импортирует как именованные, так и безымянные экспортируемые элементы, присваивая последнему имя `coolFunction`:

```
import coolFunction, {taxCode} from 'my_module';
coolFunction();
```

Обратите внимание: имя `coolFunction`, в отличие от имени `taxCode`, в фигурные скобки не заключается. Сценарий, импортирующий класс, переменную или функцию, которые экспортируются с указанием ключевого слова, должен давать им имена без применения каких-либо специальных ключевых слов:

```
import aVeryCoolFunction, {taxCode} from 'my_module';
aVeryCoolFunction();
```

Но чтобы предоставить псевдоним именованным экспортируемым элементам, нужно воспользоваться кодом, подобным следующему:

```
import coolFunction, {taxCode as taxCode2016} from 'my_module';
```

Инструкции `import` не приводят к копированию экспортируемого кода. Они служат в качестве ссылок. Сценарий, импортирующий модули или элементы, не может их изменять, и если значения в импортируемых модулях изменяются, то новые значения тут же оказывают влияние на все места, куда они были импортированы.

A.9.2. Загрузка модулей в ES6

В ранних проектах спецификации ES6 определен динамический загрузчик модулей под названием System, но в окончательную версию спецификации он не попал. В будущем объект `System` будет естественным образом реализован браузерами как загрузчик на основе промисов, который может быть использован следующим образом:

```
System.import('someModule')
      .then (function(module){
           module.doSomething();
        })
      .catch (function(error){
           // здесь выполняется обработка ошибок
        })
        ;
```

Поскольку пока объект `System` еще не реализован ни в одном браузере, мы воспользуемся полифиллом. Одним из полифиллов System является ES6 Module Loader, а другим — SystemJS.

ПРИМЕЧАНИЕ

В то время как ES6-module-loader.js является полифиллом для объекта System, загружающим только модули ES6, универсальный загрузчик SystemJS поддерживает не только модули ES6, но и модули AMD и CommonJS. В данной книге, начиная с главы 3, повсюду используется SystemJS (за исключением главы 10, в которой применяется загрузчик из Webpack). SystemJS позволяет загружать код JavaScript, а также файлы CSS и HTML в динамическом режиме.

Полифилл для ES6 Module Loader доступен в GitHub на https://github.com/ModuleLoader/es-module-loader. Вы можете скачать и распаковать этот загрузчик, скопировать файл `ES6-module-loader.js` в каталог вашего проекта и включить его в ваш HTML-файл перед сценариями вашего приложения:

```
<script src="es6-module-loader.js"></script>
<script src="my_app.js"></script>
```

Чтобы гарантировать работу сценария ES6 на всех браузерах, нужно транспилировать его в ES5. Это можно сделать либо заранее, в качестве части процесса сборки, либо динамически в браузере. Мы покажем вам последний вариант, используя компилятор Traceur.

В HTML-файл нужно включить транспилятор, загрузчик модулей и ваш сценарий (или сценарии). Можно загрузить сценарий Traceur в ваш локальный каталог или предоставить прямую ссылку на него:

```
<script src="https://google.github.io/traceur-compiler/bin/traceur.js">
</script>
<script src="es6-module-loader.js"></script>
<script src="my-es6-app.js"></script>
```

Рассмотрим простое приложение интернет-магазина, имеющее загружаемые по требованию модули доставки товаров и выставления счетов. Приложение состоит из одного HTML-файла и двух модулей. В HTML-файле имеется одна кнопка с надписью Load the Shipping Module (Загрузить модуль доставки). Когда пользователь ее нажимает, приложение должно загрузить модуль доставки и воспользоваться им, а он, в свою очередь, зависит от модуля выставления счетов. Модуль доставки имеет следующий вид (листинг А.16).

Листинг А.16. Содержимое файла shipping.js

```
import {processPayment} from 'billing';
export function ship() {
    processPayment();
    console.log("Shipping products...");
}
```

```
function calculateShippingCost(){
    console.log("Calculating shipping cost");
}
```

Функция `ship()` может вызываться внешними сценариями, а функция `calculateShippingCost()` является закрытой. Модуль доставки начинается с инструкции `import`, поэтому может вызвать функцию `processPayment()` из показанного далее модуля выставления счетов (листинг А.17).

Листинг А.17. Содержимое файла billing.js

```
function validateBillingInfo() {
    console.log("Validating billing info...");
}
export function processPayment(){
    console.log("processing payment...");
}
```

В модуле выставления счетов имеется также открытая функция `processPayment()` и закрытая функция `validateBillingInfo()`.

HTML-файл включает одну кнопку с обработчиком события щелчка, загружающим модуль доставки с помощью `System.import()` из ES6-module-loader (листинг А.18).

Листинг А.18. Содержимое файла moduleLoader.html

```
<!DOCTYPE html>
<html>
<head>
    <title>modules.html</title>
    <script src="https://google.github.io/traceur-compiler/bin/traceur.js">
    </script>
    <script src="es6-module-loader.js"></script>
</head>
<body>

  <button id="shippingBtn">Load the Shipping Module</button>

<script type="module">

let btn = document.querySelector('#shippingBtn');
btn.addEventListener('click', () => {

    System.import('shipping')
        .then(function(module) {
            console.log("Shipping module Loaded. ", module);

            module.ship();

            module.calculateShippingCost(); ◄── Выдает ошибку
        })
        .catch(function(err){
```

```
            console.log("In error handler", err);
        });
});

</script>

</body>
</html>
```

Метод `System.import()` возвращает ES6-объект `Promise`, а когда модуль загружен, выполняется функция, указанная в `then()`. В случае ошибки управление передается функции `catch()`.

В `then()` в консоли выводится сообщение, и из модуля доставки вызывается функция `ship()`, запускающая из модуля выставления счетов функцию `processPayment()`. После этого предпринимается попытка вызвать имеющуюся в модуле функцию `calculateShippingCost()`, которая заканчивается выдачей ошибки, поскольку эта функция не была экспортирована и осталась закрытой.

СОВЕТ

Если используется Traceur и в HTML-файле имеется встроенный сценарий, то, чтобы гарантировать транспиляцию кода с помощью Traceur в ES5, задействуйте атрибут type="module". Без него в браузерах, не поддерживающих ключевое слово let и стрелочные функции, этот сценарий работать не будет.

Чтобы запустить приведенный пример на своем компьютере, вам необходимо наличие Node.js с установленным средством npm. Затем загрузите и установите в любой каталог загрузчик модулей ES6-module-loader, введя следующую команду npm:

```
npm install ES6-module-loader
```

Затем создайте папку приложения и скопируйте в нее файл `ES6-module-loader.js` (это минимизированная версия загрузчика, загруженная с помощью npm). У приложения, используемого в качестве примера, имеется три дополнительных файла, показанных в листингах А.16–А.18. Чтобы ничего не усложнять, все три файла нужно хранить в одной папке.

ПРИМЕЧАНИЕ

Чтобы увидеть этот код в действии, вам нужно его «подать», используя веб-сервер. Можно установить базовый HTTP-сервер, подобный live-серверу, применив пояснения, которые были даны в подразделе 2.1.4.

Мы запускали `moduleLoader.html` в Google Chrome и открывали панель Developer Tools (Инструменты разработчика). На рис. А.6 показано, как выглядит окно этого браузера после нажатия кнопки Load the Shipping Module (Загрузить модуль доставки).

Обратите внимание на вкладку XHR в середине окна. HTML-страница загружает `shipping.js` и `billing.js` только после того, как пользователь нажмет кнопку.

Эти файлы невелики по размеру (440 и 387 байт, включая объекты HTTP-ответа), и выполнение дополнительного сетевого вызова для их получения представляется неким излишеством. Но если приложение состоит из десяти модулей по 500 Кбайт каждый, то в его разбиении на модули и применении ленивой загрузки появляется вполне определенный смысл.

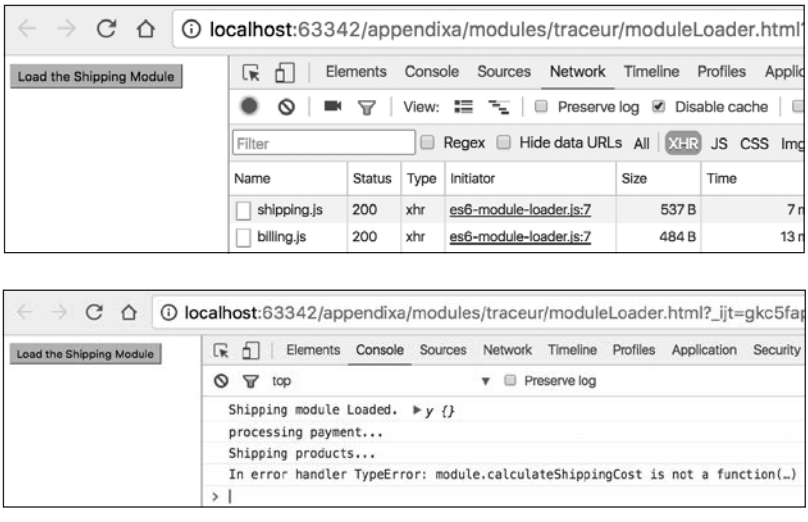


Рис. А.6. Использование загрузчика модулей ES6-module-loader

В нижней части рисунка на вкладке Console (Консоль) можно увидеть сообщение из сценария в `moduleLoader`, свидетельствующее о загрузке модуля доставки. Затем сценарий вызывает функцию `ship()` из модуля доставки и выдает, как и ожидалось, ошибку при попытке вызова функции `calculateShippingCost()`.

ПРИМЕЧАНИЕ

Целью этого приложения было познакомить вас с синтаксисом ES6. Для его углубленного изучения прочитайте книгу Exploring ES6, написанную Акселем Раушмайером (Axel Rauschmayer) (http://exploringjs.com/es6/). Эрик Дуглас (Eric Douglas) собирает на GitHub различные обучающие ресурсы, имеющие отношение к ES6, получить доступ к которым можно на https://github.com/ericdouglas/ES6-Learning.

Приложение Б. TypeScript в качестве языка для приложений Angular

Интересно, а почему бы просто не вести разработку на JavaScript? Зачем использовать другие языки программирования, если уже есть язык JavaScript? Вам ведь не попадались статьи о языках для разработки приложений Java или C#?

Дело в том, что разработка на JavaScript не отличается высокой продуктивностью. Предположим, функция ожидает в качестве аргумента строковое значение, но разработчик ошибочно вызывает ее с передачей числового значения. При работе с JavaScript эта ошибка может быть обнаружена только в ходе выполнения сценария. Компиляторы Java или C# не станут даже компилировать код, в котором имеется несоответствие типов, но интерпретатор JavaScript такой код пропускает, поскольку относится к динамически типизированным языкам.

Несмотря на то что движки JavaScript проделывают вполне достойную уважения работу, выстраивая догадки о типах переменных по их значениям, средства разработки располагают весьма ограниченными возможностями по оказанию помощи, ничего не зная о типах. В приложениях, имеющих средний и большой объем кода, этот недостаток JavaScript снижает продуктивность труда разработчиков программных продуктов.

Для более крупных проектов особую важность приобретают качественная контекстно зависимая помощь интегрированной среды разработки (IDE) и поддержка реструктуризации. Переименование во всех местах появлений той или иной переменной или функции в статически типизированных языках выполняется IDE-средами в доли секунды, даже в проектах, состоящих из тысяч строк кода, но сказать такое же о JavaScript, языке, не поддерживающем типы, невозможно. IDE-среды могут помочь с реструктуризацией гораздо эффективнее, когда известны типы переменных.

Чтобы повысить продуктивность работы, можно рассмотреть возможность ведения разработки в статически типизированных языках с последующей конвертацией кода в JavaScript для развертывания приложения. В настоящее время существует дюжина языков, компилируемых в JavaScript (их перечень можно найти на GitHub (https://github.com/jashkenas/coffeescript/wiki/List-of-languages-that-compile-to-JS). Наиболее популярными из них являются TypeScript (www.typescriptlang.org), CoffeeScript (http://coffeescript.org) и Dart (www.dartlang.org).

А почему бы не воспользоваться языком DART?

Мы уделили работе с языком Dart немало времени, и он нам понравился, но у него все же есть ряд недостатков:

- степень взаимодействия с JavaScript-библиотеками сторонних разработчиков невелика;
- вести разработку на Dart можно только на специализированных версиях браузера Chrome (Dartium), поставляемого вместе с виртуальной машиной Dart VM. Другие браузеры такой возможности не предоставляют;
- код, создаваемый JavaScript, человеку читать сложно;
- сообщество разработчиков, пользующихся языком Dart, сравнительно мало.

Фреймворк Angular написан на TypeScript, и в данном приложении будет рассмотрен синтаксис именно этого языка. Все примеры кода, приводимые в данной книге, написаны на TypeScript. Кроме того, мы показали вам, как превратить код TypeScript в его JavaScript-версию, чтобы он мог выполняться любым браузером или автономным движком JavaScript.

Б.1. Зачем создавать Angular-приложения на TypeScript

Можно создавать приложения на ES6 (и даже на ES5), но мы воспользовались TypeScript как значительно более продуктивным способом написания кода JavaScript, и вот почему.

- TypeScript поддерживает типы. Он позволяет компилятору TypeScript помогать вам искать и устранять множество ошибок в ходе разработки, даже перед запуском приложения.
- Одним из основных преимуществ TypeScript является хорошая поддержка IDE-среды. При ошибке, допущенной в имени функции или переменной, код отображается красным цветом. При передаче функции неверного количества параметров (или неверных типов) все, что не подходит, показывается красным. IDE-среды также предоставляют великолепную контекстно зависимую помощь. Код TypeScript может быть реструктурирован IDE-средами, а код JavaScript должен реструктурироваться вручную. Если нужно исследовать новую библиотеку, то следует просто установить ее файл определения типов, и IDE-среда проинформирует о наличии доступных API, избавляя от необходимости читать ее документацию из какого-либо другого источника.
- В пакет Angular входят файлы определения типов, поэтому IDE-среды проверяют соответствие типов при использовании API Angular и тут же предлагают контекстно зависимую помощь.
- TypeScript следует положениями спецификаций ECMAScript 6 и 7 и добавляет к ним типы, интерфейсы, декораторы, переменные элементов класса

(поля), обобщения и ключевые слова `public` и `private`. Предстоящие выпуски TypeScript будут поддерживать отсутствующие функции ES6 и реализовывать функции ES7 (см. «Дорожную карту» TypeScript на GitHub на https://github.com/Microsoft/TypeScript/wiki/Roadmap).

- Интерфейсы TypeScript позволяют объявлять пользовательские типы, используемые в вашем приложении. Интерфейсы помогают избегать ошибки в ходе компиляции, вызванные применением в вашем приложении объектов неверных типов.
- Созданный код JavaScript легко читается и выглядит как код, написанный вручную.
- Большинство примеров кода в документации по Angular, статьях и блогах дается в TypeScript (см. https://angular.io/docs).

Б.2. Роль транспиляторов

Браузеры понимают только язык JavaScript. В случае если исходный код написан на TypeScript, прежде чем запускать его в браузере или автономном движке JavaScript, нужно его транспилировать в JavaScript.

Транспиляция означает преобразование исходного кода программы на одном из языков в исходный код программы на другом языке. Многие разработчики отдают предпочтение слову «*компиляция*», поэтому такие фразы, как «компилятор TypeScript» и «компиляция TypeScript в JavaScript», тоже имеют право на существование.

На рис. Б.1 показан снимок экрана с кодом TypeScript слева и его эквивалентом в ES5-версии JavaScript, созданным транспилятором TypeScript. В TypeScript объявляется переменная `foo` типа `string`, а в транспилированной версии информация о типе отсутствует. В TypeScript объявляется класс `Bar`, который был транспилирован в похожий на класс шаблон в синтаксисе ES5. Если в качестве цели транспиляции был бы указан ES6, то созданный код JavaScript выглядел бы иначе.

```
1  var foo: string;
2
3  class Bar{
4
5  }
```

```
1  var foo;
2  var Bar = (function () {
3      function Bar() {
4      }
5      return Bar;
6  })();
7
```

Рис. Б.1. Транспиляция TypeScript в ES5

Сочетание Angular со статически типизированным TypeScript упрощает разработку веб-приложений среднего и большого объема. Качественное инструментальное оснащение и анализатор статических типов существенно сокращают количество ошибок, выявляемых в ходе выполнения программы, и уменьшают время вывода программного продукта на рынок. Когда работа над

проектом завершится, в вашем Angular-приложении будет большой объем кода на JavaScript. И хотя разработка на TypeScript потребует написания большего объема кода, вы станете пожинать плоды за счет экономии времени на тестировании и реорганизации, а также на сведении к минимуму количества ошибок, выявляемых при выполнении программы.

Б.3. Начало работы с TypeScript

У компании Microsoft имеется TypeScript с открытым кодом, и она разместила репозиторий TypeScript на GitHub (https://github.com/Microsoft/TypeScript/wiki/Roadmap). Компилятор TypeScript можно установить, используя npm или загрузив его с www.typescriptlang.org. На сайте TypeScript также имеется встроенный (интерактивный) компилятор TypeScript, в который можно вводить код TypeScript и компилировать его в код JavaScript в интерактивном режиме, как показано на рис. Б.2.

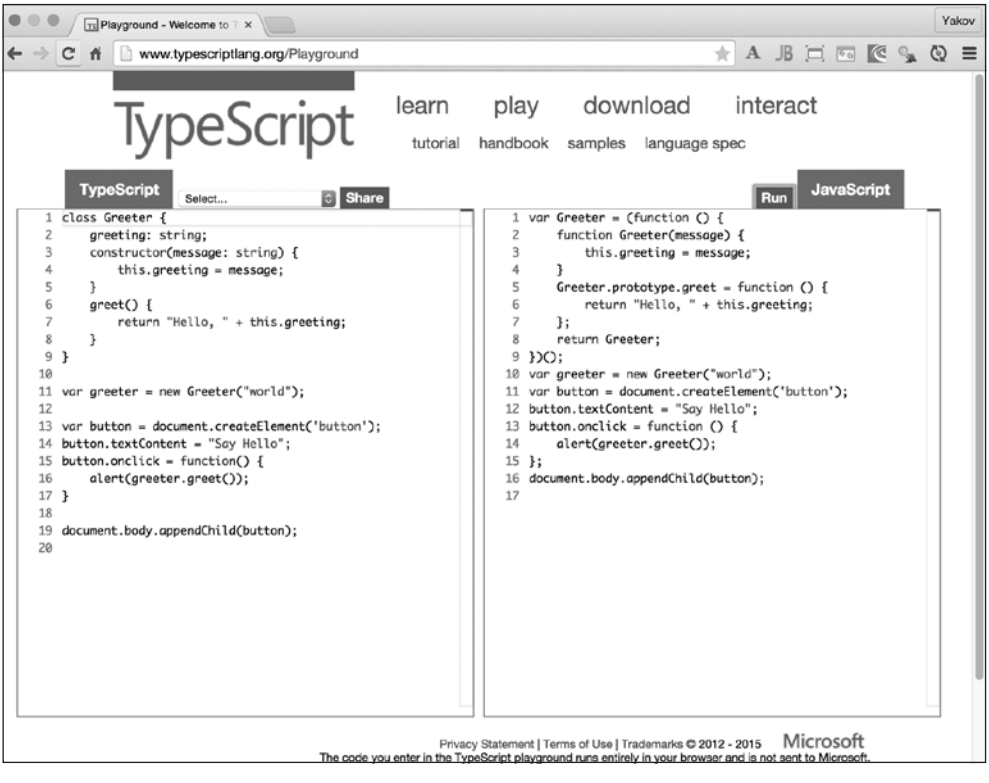


Рис. Б.2. Применение интерактивной среды TypeScript

В интерактивной среде TypeScript код на этом языке вводится в левой части окна, а его JavaScript-версия появляется в его правой части. Для запуска транспилированного кода нужно нажать кнопку Run (Запуск) (консольный вывод, про-

изведенный вашим кодом, если таковой имеется, будет доступен для просмотра после открытия в браузере панели Developer Tools (Инструменты разработчика)).

Интерактивного средства будет достаточно, чтобы изучить синтаксис языка, но для настоящей продуктивной разработки нужна более подходящая оснастка. Можно остановиться на использовании IDE-среды или текстового редактора, но для разработки будет не обойтись без локально установленного компилятора TypeScript.

Б.3.1. Установка и применение компилятора TypeScript

Сам компилятор TypeScript написан на TypeScript. Для его установки нужно воспользоваться имеющимся в Node.js диспетчером пакетов npm. Если Node отсутствует, то его нужно загрузить с https://nodejs.org/en/ и установить на компьютер. Система Node.js поставляется с npm, который будет применен для установки не только компилятора TypeScript, но и многих других средств разработки, упоминаемых в данной книге.

Для глобальной установки компилятора TypeScript нужно запустить в окне команд или терминала следующую npm-команду:

```
npm install -g typescript
```

Ключ `-g` приведет к глобальной установке компилятора TypeScript на вашем компьютере, и он станет доступен из приглашения командной строки всем вашим проектам. Для разработки приложений Angular загрузите самую последнюю версию компилятора TypeScript (при написании этой книги использовалась версия 2.0).

Для проверки версии вашего компилятора TypeScript запустите следующую команду:

```
tsc --version
```

Созданный на TypeScript код нужно транспилировать в JavaScript, чтобы браузер мог его выполнить. Код на TypeScript сохраняется в файлах с расширением `.ts`. Предположим, вы написали сценарий, который сохранен в файле `main.ts`. Тогда следующая команда приведет к транспиляции `main.ts` в `main.js`:

```
tsc main.ts
```

Можно также создавать файлы отображения исходного кода, проецирующие исходный код TypeScript на созданный код JavaScript. Имея эти отображения, можно устанавливать контрольные точки в вашем коде TypeScript в момент запуска сценария в браузере, даже притом, что последний будет выполнять код JavaScript. Для компиляции `main.ts` в `main.js` с попутным созданием файла отображения исходного кода `main.map` нужно запустить следующую команду:

```
tsc --sourcemap main.ts
```

На рис. Б.3 показан снимок экрана, полученный при отладке, проводимой на панели Developer Tools (Инструменты разработчика) браузера Chrome. Обратите внимание на контрольную точку в строке 15. На вкладке Sources (Исходные коды)

этой панели можно найти ваш файл TypeScript, поместить в коде контрольную точку и отслеживать значения переменных в правой стороне экрана.

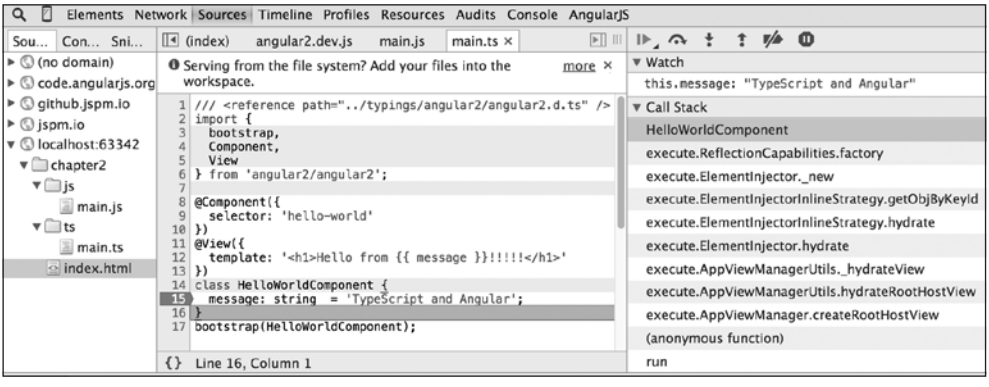


Рис. Б.3. Отладка TypeScript на панели Developer Tools (Инструменты разработчика)

Чтобы получился корректный код JavaScript, в ходе компиляции компилятор TypeScript удаляет из создаваемого кода все типы TypeScript, интерфейсы и ключевые слова. Предоставляя компилятору те или иные ключи, можно создавать код JavaScript, совместимый с синтаксисом ES3, ES5 или ES6. В настоящий момент по умолчанию создается код, совместимый с ES3. А вот как транспилировать код в синтаксис, совместимый с ES5:

```
tsc --t ES5 main.ts
```

Транспиляция TypeScript в браузере

В ходе разработки мы задействуем локально установленный компилятор tsc, но транспиляцию можно также выполнить либо на сервере в ходе разработки, либо динамически, когда браузер загружает ваше приложение. При написании книги мы применяли библиотеку SystemJS, внутри которой для транспиляции и динамической загрузки модулей приложения используется tsc.

Имейте в виду, что динамическая транспиляция в браузере может приводить к задержкам в отображении содержимого вашего приложения на устройствах пользователя. Если для загрузки и транспиляции вашего кода в браузер используется SystemJS, то отображения исходного кода будут создаваться по умолчанию.

При необходимости откомпилировать код в памяти без создания выходных файлов с расширением `.js` компилятор tsc нужно запускать с ключом `--noEmit`. Мы часто задействуем этот ключ в режиме разработки, поскольку нам нужен только выполняемый код JavaScript в памяти браузера.

Компилятор TypeScript можно запускать в режиме отслеживания, предоставив для этого ключ `-w`. В данном режиме любое изменение и сохранение вашего кода будет приводить к его автоматической транспиляции в соответствующие файлы

JavaScript. Чтобы скомпилировать и отслеживать все файлы с расширением `.ts`, запустите следующую команду:

```
tsc -w *.ts
```

Компилятор скомпилирует все файлы TypeScript, выведет сообщения об ошибках (если таковые обнаружатся) на консоль и продолжит отслеживать изменения в файлах. Как только файл будет изменен, tsc тут же его перекомпилирует.

ПРИМЕЧАНИЕ

Обычно мы не используем среду IDE для компиляции TypeScript. Задействуется либо SystemJS с компилятором, работающим в среде браузера, либо упаковщик (Webpack), применяющий для компиляции специальный загрузчик TypeScript. Мы используем анализатор кода TypeScript, предоставляемый IDE-средами для выделения ошибок, и браузер для отладки TypeScript.

Компилятор TypeScript позволяет заранее сконфигурировать процесс компиляции (указать каталоги источника и получателя, создать отображения исходного кода и т. д.). Присутствие в каталоге проекта конфигурационного файла `tsconfig.json` означает следующее: ввод `tsc` приведет к тому, что компилятор считает все ключи из этого файла. Пример файла `tsconfig.json` показан в листинге Б.1.

Листинг Б.1. Содержимое файла tsconfig.json

```
{
    "compilerOptions": {
      "target": "es5",
      "module": "commonjs",
      "emitDecoratorMetadata": true,
      "experimentalDecorators": true,
      "rootDir": ".",
      "outDir": "./js"
    }
}
```

Этот файл конфигурации настраивает tsc на транспилирование кода в синтаксис ES5. Создаваемые файлы JavaScript будут размещены в каталоге `js`. Файл `tsconfig.json` может включать раздел `files`, в котором перечисляются файлы, подлежащие компиляции TypeScript. В листинг Б.1 данный перечень не включен, поскольку в нем используется ключ `rootDir` для запроса компиляции всех файлов, начиная с корневого каталога проекта.

Если необходимо исключить некоторые файлы проекта из компиляции, то к `tsconfig.json` следует добавить свойство `exclude`. Исключить все содержимое каталога `node_modules` можно следующим образом:

```
"exclude": [
    "node_modules"
]
```

Дополнительные сведения о конфигурировании процесса компиляции и ключах компилятора TypeScript можно найти в документации TypeScript (http://www.typescriptlang.org/docs/handbook/compiler-options.html).

ПРИМЕЧАНИЕ

В большинстве примеров Angular, приведенных в данной книге, с классами или элементами классов используются аннотации (также называемые декораторами), например, @Component и @Input. Аннотации являются способом добавления метаданных к аннотируемым классам или их элементам. Подробности можно найти во врезке «Что такое метаданные» в подразделе 2.1.1.

Б.4. TypeScript как расширенная версия JavaScript

В TypeScript полностью поддерживаются ES5 и основная часть синтаксиса ES6. Можно просто изменить расширение файла с кодом JavaScript с `.js` на `.ts`, и он станет файлом с вполне допустимым кодом TypeScript. До сих пор нам попадались только два исключения, касающиеся обработки необязательных параметров функций и присваивания значения литералу объекта.

В JavaScript, даже если функция объявлена с двумя параметрами, ее можно вызвать, предоставив только один параметр, а в TypeScript, чтобы сделать параметр необязательным, к его имени нужно добавить знак вопроса. В JavaScript можно инициализировать переменную, указав пустой литерал объекта, и тут же прикрепить свойство, используя запись через точку, а в TypeScript придется задействовать квадратные скобки.

Но эти различия минимальны. Гораздо важнее то, что, являясь расширенной версией JavaScript, TypeScript добавляет к JavaScript ряд весьма полезных функций, который мы и рассмотрим далее.

СОВЕТ

Если вы находитесь на полпути преобразования проекта JavaScript в TypeScript, то можете воспользоваться имеющимся в компиляторе tsc ключом --allowJs. Компилятор TypeScript проверит входные файлы с расширением .js на синтаксические ошибки и выдаст допустимый результат, основываясь на ключах tsc --target и --module. Его можно объединить с другими файлами с расширением .ts. Отображения исходного кода по-прежнему создаются для файлов .js, точно так же, как и для файлов .ts.

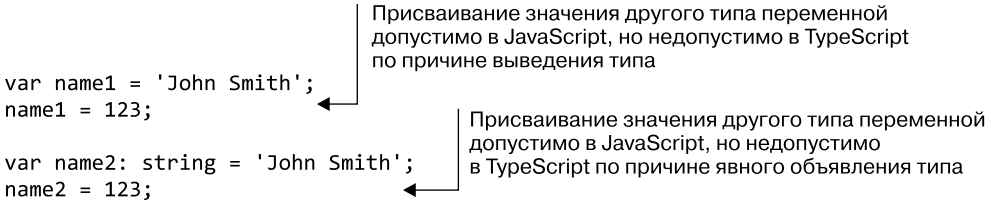
Б.5. Необязательные типы

Можно объявлять переменные и предоставлять типы для всех этих переменных или только для некоторых из них. Синтаксис TypeScript допускает обе строки кода, представленные ниже:

```
var name1 = 'John Smith';
var name2: string = 'John Smith';
```

При использовании типов транспилятор TypeScript может обнаружить несоответствие типов в ходе разработки, а IDE-среды станут предлагать варианты завершения кода и поддержку реструктуризации. Это повысит производительность вашей работы над любым достаточно большим по объему проектом. Даже если не применять типы в объявлениях, TypeScript сам выведет тип на основе присвоенного значения и потом все равно станет проверять соответствие типов. Данная особенность называется *выведением типа*.

В следующем фрагменте кода TypeScript показана невозможность присвоить числовое значение переменной `name1`, которая должна была быть строковой переменной, даже притом, что изначально она объявлена без указания типа (с помощью синтаксиса JavaScript). После инициализации этой переменной строковым значением выведение типов не позволит присвоить `name1` числовое значение. Те же правила применимы и к переменной `name2`, объявленной с явным указанием типа:



В TypeScript допускается объявление типизированных переменных, параметров функций и возвращаемых значений. Для объявления основных типов используются четыре ключевых слова: `number`, `boolean`, `string` и `void`. Последнее показывает в объявлении функции отсутствие возвращаемого значения. Как и в JavaScript, переменной может быть значение типа `null` или `undefined`.

Рассмотрим некоторые примеры переменных, объявленных с явным указанием типов:

```
var salary: number;
var name: string = "Alex";
var isValid: boolean;
var customerName: string = null;
```

Все эти типы являются подтипами типа `any`. Если при объявлении переменной или аргумента функции тип не указан, то компилятор TypeScript сделает предположение, что у него имеется тип `any`; это позволит присвоить данной переменной или аргументу функции любое значение.

Можно также воспользоваться явным объявлением переменной с указанием `any` в качестве ее типа. В таком случае выведение типа применено не будет. Допустимы оба следующих объявления:

```
var name2: any = 'John Smith';
name2 = 123;
```

Если переменные объявлены с явным указанием типов, то компилятор проверит их значения, чтобы убедиться в их соответствии объявлениям. В TypeScript включены и другие типы, используемые при взаимодействии с браузером, например `HTMLElement` и `Document`.

Если вы определяете класс или интерфейс, то можете применять его в объявлениях переменных в качестве пользовательского типа. К классам и интерфейсам мы еще вернемся, но сначала познакомимся с функциями TypeScript, являющимися в JavaScript наиболее востребованными конструкциями.

Б.5.1. Функции

Функции TypeScript (и функциональные выражения) похожи на функции JavaScript, но при этом имеется возможность явного объявления типов параметров и возвращаемых значений. Напишем функцию JavaScript, вычисляющую размер налога (листинг Б.2). У нее будет три параметра, и она станет вычислять налог на основе штата проживания, суммы дохода и количества иждивенцев. Для каждого иждивенца в зависимости от штата проживания имеется право на вычет из налогооблагаемой суммы в размере $500 или 300.

Листинг Б.2. Вычисление суммы налога с помощью JavaScript

```
function calcTax(state, income, dependents) {
    if (state == 'NY') {
        return income * 0.06 - dependents * 500;
    } else if (state == 'NJ') {
        return income * 0.05 - dependents * 300;
    }
}
```

Предположим, что налогоплательщик с доходом $50 000 живет в штате Нью-Джерси и имеет на иждивении двух человек. Вызовем `calcTax()`:

```
var tax = calcTax('NJ', 50000, 2);
```

Переменная `tax` получает значение `1900`, не вызывающее возражений. Даже притом, что в `calcTax()` для параметров функции не объявляются никакие типы, их можно вывести на основе имен параметров.

Теперь вызовем функцию неподобающим образом, передав для количества иждивенцев строковое значение:

```
var tax = calcTax('NJ', 50000, 'two');
```

Выявить проблему до вызова этой функции невозможно. Переменная `tax` будет иметь значение `NaN` (не число). Ошибка вкралась только по причине невозможности явного указания типов параметров. Перепишем данную функцию на TypeScript, объявив типы для параметров и возвращаемого значения (листинг Б.3).

Листинг Б.3. Вычисление суммы налога с помощью TypeScript

```
function calcTax(state: string, income: number, dependents: number): number{
    if (state == 'NY'){
        return income*0.06 - dependents*500;
    } else if (state=='NJ'){
        return income*0.05 - dependents*300;
    }
}
```

Теперь допустить такую же ошибку и передать строковое значение для количества иждивенцев невозможно:

```
var tax: number = calcTax('NJ', 50000, 'two');
```

Компилятор TypeScript выдаст ошибку со следующим сообщением: Argument of type 'string' is not assignable to parameter of type 'number'. Более того, для возвращаемого значения функции объявлен тип `number`, который не позволит допустить еще одну ошибку и присвоить результат вычисления налога нечисловой переменной:

```
var tax: string = calcTax('NJ', 50000, 'two');
```

Компилятор выявит эту ошибку и выдаст сообщение: The type 'number' is not assignable to type 'string': var tax: string. Такая проверка соответствия типов в ходе компиляции позволит вам сэкономить уйму времени при разработке любого проекта.

Б.5.2. Параметры по умолчанию

При объявлении функции можно указать значения параметров по умолчанию. Единственным ограничением является то, что параметры со значениями по умолчанию не могут иметь после себя обязательные параметры. В листинге Б.3, чтобы предоставить NY в качестве значения по умолчанию для параметра `state`, объявить это следующим образом невозможно:

```
function calcTax(state: string = 'NY', income: number, dependents: number):
➥ number{
    // сюда помещается код }
```

Чтобы гарантировать отсутствие обязательных параметров после параметра по умолчанию, нужно изменить порядок следования параметров:

```
function calcTax(income: number, dependents: number, state: string = 'NY'):
➥ number{
    // сюда помещается код
}
```

В теле `calcTax()` не нужно изменять ни одной строки кода. Теперь можно совершенно свободно вызвать эту функцию либо с двумя, либо с тремя параметрами:

```
var tax: number = calcTax(50000, 2);
// или
var tax: number = calcTax(50000, 2, 'NY');
```

Результат обоих вызовов будет одинаковым.

Б.5.3. Необязательные параметры

В TypeScript можно просто пометить параметры функции как необязательные, добавив к имени параметра знак вопроса. Единственным ограничением является то, что дополнительные параметры должны указываться в объявлении функции последними. При написании кода для функций с необязательными параметрами необходимо реализовать логику приложения, обрабатывающую те случаи, когда необязательные параметры не предоставлены.

Изменим функцию вычисления суммы налога: если количество иждивенцев не указано, то вычеты к вычислению налога применяться не будут (листинг Б.4).

Листинг Б.4. Измененное вычисление суммы налога с помощью TypeScript

```
function calcTax(income: number, state: string = 'NY', dependents?: number):
➥ number{

    var deduction: number;

    if (dependents) {
        deduction = dependents*500;
    }else {
        deduction = 0;
    }

    if (state == 'NY'){
        return income*0.06 - deduction;
    } else if (state=='NJ'){
        return income*0.05 - deduction;
    }
}

var tax: number = calcTax(50000, 'NJ', 3);
console.log("Your tax is " + tax);

var tax: number = calcTax(50000);
console.log("Your tax is " + tax);
```

Обрабатывает дополнительное значение в dependents

Обратите внимание на знак вопроса в `dependents?: number`. Теперь функция проверяет, предоставлено ли значение для количества иждивенцев. Если нет, то переменной `deduction` присваивается значение `0`, в противном случае из каждого иждивенца вычитается `500`.

Запуск кода листинга Б.4 выдаст следующий результат:

```
Your tax is 1000
Your tax is 3000
```

Б.5.4. Выражения стрелочных функций

В TypeScript поддерживается упрощенный синтаксис использования в выражениях безымянных функций. При этом исключается необходимость применения ключевого слова `function`, и для отделения параметров функции от ее тела служит знак жирной стрелки (`=>`). В TypeScript для стрелочных функций поддерживается синтаксис ES6 (более подробно данные функции рассмотрены в приложении А). В некоторых других языках программирования стрелочные функции известны как *лямбда-выражения*.

Рассмотрим простейший пример стрелочной функции с телом, умещающимся в одной строке:

```
var getName = () => 'John Smith';
console.log(getName());
```

Пустые круглые скобки обозначают отсутствие в предшествующей стрелочной функции параметров. Для стрелочного выражения, умещающегося в одной строке, не нужны ни фигурные скобки, ни явно указываемая инструкция `return`, и предыдущий фрагмент кода выведет в консоли `John Smith`. Если ввести этот код в интерактивную среду TypeScript, то она преобразует его в следующий код ES5:

```
var getName = function () { return 'John Smith'; };
console.log(getName());
```

Если тело стрелочной функции состоит из нескольких строк, то его придется заключить в фигурные скобки и воспользоваться инструкцией `return`. В следующем фрагменте кода происходит преобразование конкретно заданного строкового значения в строку с символами в верхнем регистре, и в консоли выводится `PETER LUGER`:

```
var getNameUpper = () => {
  var name = 'Peter Luger'.toUpperCase();
  return name;
}
console.log(getNameUpper());
```

Кроме предоставления более лаконичного синтаксиса, выражения стрелочных функций устраняют небезызвестную путаницу с ключевым словом `this`. Если в JavaScript оно используется в функции, то может не указывать на объект, из которого была вызвана данная функция. Это может привести к ошибкам в ходе выполнения и потребовать дополнительного времени на отладку. Рассмотрим пример.

В листинге Б.5 имеется две функции: `StockQuoteGeneratorArrow()` и `StockQuoteGeneratorAnonymous()`. Каждую секунду они обе вызывают `Math.random()` для выдачи произвольной цены на акцию, символ которой предоставлен в качестве параметра. Внутри `StockQuoteGeneratorArrow()` используется синтаксис стрелочной функции, предоставляющей аргумент для `setInterval()`, а внутри `StockQuoteGeneratorAnonymous()` применяется безымянная функция.

Листинг Б.5. Использование выражения стрелочной функции

```
Присваивание символа акции this.symbol

function StockQuoteGeneratorArrow(symbol: string){

    this.symbol = symbol;

    setInterval(() => {
        console.log("StockQuoteGeneratorArrow. The price quote for " +
        ➥ this.symbol
            + " is " + Math.random());
    }, 1000);
}

var stockQuoteGeneratorArrow = new StockQuoteGeneratorArrow("IBM");

function StockQuoteGeneratorAnonymous(symbol: string){
```

Использование стрелочной функции в качестве аргумента setInterval() для вызова ее каждую секунду (каждые 1000 миллисекунд)

```
        this.symbol = symbol;
    }

    setInterval(function () {
        console.log(" StockQuoteGeneratorAnonymous.The price quote for " +
        ➥ this.symbol
            + " is " + Math.random());
    }, 1000);
}

var stockQuoteGeneratorAnonymous = new StockQuoteGeneratorAnonymous("IBM");
```

Присваивание символа акции this.symbol

Применение анонимной функции в качестве аргумента setInterval()

В обоих случаях символ акции (`IBM`) присваивается переменной `symbol` объекта `this`, но при использовании стрелочной функции ссылка на экземпляр функции-конструктора `StockQuoteGeneratorArrow()` автоматически сохраняется в отдельной переменной. При ссылке на `this.symbol` из стрелочной функции она правильно находит эту переменную и задействует в выводе в консоли `IBM`.

Но когда в браузере вызывается безымянная функция, `this` указывает на глобальный объект `Window`, у которого нет свойства `symbol`. Запуск этого кода в браузере приведет к выводу каждую секунду информации, подобной представленной ниже:

```
StockQuoteGeneratorArrow. The price quote for IBM is 0.2998261866159737
StockQuoteGeneratorAnonymous.The price quote for undefined is
➥ 0.9333276399411261
```

Как видите, при использовании стрелочной функции она распознает `IBM` в качестве символа акции, а в безымянной функции получается неопределенность (`undefined`).

ПРИМЕЧАНИЕ

TypeScript заменяет this в выражении стрелочной функции ссылкой на this из внешней области видимости путем передачи в ссылке. Дело в том, что код в стрелочной функции в StockQuoteGeneratorArrow() правильно видит this.symbol из внешней области видимости.

Нашей следующей темой будут классы TypeScript, но сделаем небольшую паузу и подведем итоги всему только что рассмотренному:

- код Typescript компилируется в JavaScript с помощью компилятора tsc;
- TypeScript позволяет объявлять типы переменных, параметров функций и возвращаемых значений;
- функции могут иметь параметры со значениями по умолчанию, а также необязательные параметры;
- выражения стрелочных функций предлагают более лаконичный синтаксис для объявления безымянных функций;
- выражения стрелочных функций устраняют неопределенность в использовании ссылки на объект `this`.

Перегрузка функций

Функция перегрузки в JavaScript не поддерживается, поэтому иметь несколько функций с одним и тем же именем, но с разными списками аргументов невозможно. Создатели TypeScript ввели функцию перегрузки, но поскольку код должен быть транспилирован в одну функцию JavaScript, синтаксис для перегрузки выглядит совсем не элегантно.

Можно объявить несколько сигнатур функции с одним и только одним телом, где нужно будет проверять количество и типы аргументов и выполнять соответствующие части кода:

```
function attr(name: string): string;
function attr(name: string, value: string): void;
function attr(map: any): void;
function attr(nameOrMap: any, value?: string): any {
  if (nameOrMap && typeof nameOrMap === "string") {
      // обработка варианта со строкой
  } else {
      // обработка варианта с отображением
  }
  // здесь проводится обработка значения
}
```

Б.6. Классы

Тем, у кого есть опыт программирования на Java или на C#, должно быть знакомо понятие классов и наследования в их классическом виде. В таких языках определение класса загружается в память в качестве отдельной сущности (вроде общего замысла) и совместно используется всеми экземплярами этого класса. Если класс является наследником другого класса, то объект, создаваемый в качестве его экземпляра, задействует комбинированный общий замысел обоих классов.

TypeScript — расширенная версия языка JavaScript, в котором поддерживается только *прототипное наследование*, позволяющее создавать иерархию наследования путем прикрепления одного объекта к свойству `prototype` другого объекта. В данном случае наследование *объектов* (или скорее их связь) создается динамически.

В TypeScript ключевое слово `class` является синтаксической уловкой, упрощающей программирование. В конечном счете ваши классы будут транспилированы в объекты JavaScript с прототипным наследованием. В JavaScript допускается объявление функции-конструктора и создание ее экземпляра с помощью ключевого слова `new`. В TypeScript можно также объявить класс и создать его экземпляр, используя оператор `new`.

В класс можно включать конструктор, поля (они же свойства) и методы. Объявленные свойства и методы часто называют *элементами класса*. Мы проиллюстрируем синтаксис классов TypeScript, показав ряд примеров и сравнивая их с соответствующими синтаксическими конструкциями в ES5.

Создадим простой класс `Person`, содержащий четыре свойства для хранения фамилии, имени, возраста и номера карточки социального страхования (уникального идентификатора, присваиваемого лицу, легально проживающему в Соединенных Штатах). В левой части рис. Б.4 можно увидеть код TypeScript, содержащий объявление и создающий экземпляр класса `Person`, а в его правой части — функцию-замыкание на JavaScript, созданную компилятором tsc. Создавая замыкание для функции `Person`, компилятор TypeScript обеспечивает механизм для экспонирования и скрытия элементов объекта `Person`.

```
class Person {
    firstName: string;
    lastName: string;
    age: number;
    ssn: string;
}

var p = new Person();

p.firstName = "John";
p.lastName = "Smith";
p.age = 29;
p.ssn = "123-90-4567";
```

```
var Person = (function () {
    function Person() {
    }
    return Person;
})();
var p = new Person();
p.firstName = "John";
p.lastName = "Smith";
p.age = 29;
p.ssn = "123-90-4567";

```

Рис. Б.4. Транспиляция класса TypeScript в замыкание JavaScript

В TypeScript также поддерживаются конструкторы класса, позволяющие инициализировать переменные объекта при создании его экземпляра. Конструктор класса вызывается только один раз в ходе создания объекта. В левой части рис. Б.5 показана еще одна версия класса `Person`, в которой используется ключевое слово `constructor`, позволяющее проинициализировать поля класса значениями, переданными конструктору. В правой части рисунка показана созданная ES5-версия кода.

```
class Person {
    firstName: string;
    lastName: string;
    age: number;
    ssn: string;

    constructor(firstName:string, lastName: string,
        age: number, ssn: string) {

        this.firstName = firstName;
        this.lastName;
        this.age = age;
        this.ssn = ssn;
    }
}

var p = new Person("John", "Smith", 29, "123-90-4567");
```

```
var Person = (function () {
    function Person(firstName, lastName, age, ssn) {
        this.firstName = firstName;
        this.lastName;
        this.age = age;
        this.ssn = ssn;
    }
    return Person;
})();
var p = new Person("John", "Smith", 29, "123-90-4567");

```

Рис. Б.5. Транспиляция имеющегося в TypeScript класса с конструктором

Некоторые разработчики программных средств на JavaScript могут посчитать, что от использования классов мало проку, поскольку они легко смогут запрограммировать те же самые функциональные свойства с помощью функций-конструкторов и замыканий. Но те, кто только что приступил к работе с JavaScript, сочтут синтаксис классов более легким для чтения и написания, по сравнению с функциями-конструкторами и замыканиями.

Б.6.1. Модификаторы доступа

В JavaScript невозможно объявить переменную или метод закрытыми (спрятанными от внешнего кода). Для скрытия свойства (или метода) в объекте нужно создать замыкание, которое не будет ни прикреплять это свойство к переменной `this`, ни возвращать его в инструкции `return` замыкания.

TypeScript предоставляет ключевые слова `public`, `protected` и `private`, чтобы помочь управлять доступом к элементам объектов в ходе разработки. По умолчанию все элементы класса находятся в открытом доступе (`public`) и видимы за пределами класса. Если элемент объявлен с модификатором `protected` (защищенный), то он видим в классе и его подклассах. Элемент класса, объявленный с модификатором `private` (закрытый), видим только в классе.

Применим ключевое слово `private` в целях скрыть значение свойства `ssn`, чтобы исключить к нему непосредственный доступ за пределами объекта `Person`. Мы покажем две версии объявления класса со свойствами, в которых задействованы модификаторы доступа. Более длинная версия класса имеет следующий вид (листинг Б.6).

Листинг Б.6. Использование закрытого свойства

```
class Person {
    public firstName: string;
    public lastName: string;
    public age: number;
    private _ssn: string;
    constructor(firstName:string, lastName: string, age: number, ssn: string)
{
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
        this._ssn = ssn;
    }
}
var p = new Person("John", "Smith", 29, "123-90-4567");
console.log("Last name: " + p.lastName + " SSN: " + p._ssn);
```

Обратите внимание: имя закрытой переменной начинается со знака подчеркивания: `_ssn`. Тем самым в отношении закрытых свойств соблюдается соглашение об именах.

В последней строке кода листинга Б.6 предпринимается попытка доступа к закрытому свойству `_ssn` извне, поэтому анализатор кода TypeScript выдаст ошибку

компиляции: Property 'ssn' is private and is only accessible in class 'Person'. Но пока компилятор не будет запущен с ключом `--noEmitOn-Error`, код с ошибкой все равно будет транспилирован в JavaScript:

```
var Person = (function () {
    function Person(firstName, lastName, age, _ssn) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
        this._ssn = _ssn;
    }
    return Person;
})();
var p = new Person("John", "Smith", 29, "123-90-4567");
console.log("Last name: " + p.lastName + " SSN: " + p._ssn);
```

Ключевое слово `private` придает свойство закрытости только в коде TypeScript. IDE-среды не станут показывать закрытые элементы в контекстно зависимой помощи при попытке доступа к свойствам объекта извне, но в создаваемом коде JavaScript все свойства и методы класса будут все равно рассматриваться как открытые.

TypeScript позволяет предоставлять модификаторы доступа с аргументами конструктора, как показано в следующей короткой версии класса `Person` (листинг Б.7).

Листинг Б.7. Использование модификаторов доступа

```
class Person {
    constructor(public firstName: string,
        public lastName: string, public age: number, private _ssn: string) {
    }
}
var p = new Person("John", "Smith", 29, "123-90-4567");
```

При использовании конструктора с модификаторами доступа компилятор TypeScript воспринимает модификатор в качестве инструкции по созданию и сохранению свойств класса, соответствующих аргументам конструктора. Их не нужно объявлять и инициализировать явным образом. Обе версии класса `Person`, как короткая, так и длинная, приводят к созданию одного и того же кода JavaScript.

Б.6.2. Методы

Когда функция объявляется в классе, ее называют *методом*. В JavaScript методы нужно объявлять в прототипе объекта; но в классе метод объявляется путем указания имени, за которым ставятся круглые и фигурные скобки, как это делается в других объектно-ориентированных языках.

В следующем фрагменте кода показано, как можно объявить и использовать класс `MyClass` с методом `doSomething()`, имеющим один аргумент и не имеющим возвращаемого значения (листинг Б.8).

Листинг Б.8. Создание метода

```
class MyClass{
    doSomething(howManyTimes: number): void{
        // выполнение какой-либо операции
    }
}
var mc = new MyClass();
mc.doSomething(5);
```

Статические элементы и элементы экземпляра

В коде листинга Б.8, а также в коде класса, показанного на рис. Б.4, сначала создается экземпляр класса, а затем происходит обращение к его элементам с помощью ссылочной переменной, указывающей на этот экземпляр:

```
mc.doSomething(5);
```

Если при объявлении свойства или метода применялось ключевое слово `static`, то его значения будут совместно использоваться всеми экземплярами класса, и вам не придется создавать экземпляр для доступа к статическим элементам. Вместо ссылочной переменной (такой как `mc`) нужно задействовать имя класса:

```
class MyClass{
    static doSomething(howManyTimes: number): void{
        // выполнение какой-либо операции
    }
}
MyClass.doSometing(5);
```

Если создан экземпляр класса и нужно вызвать метод класса из другого метода, объявленного в том же самом классе, то следует воспользоваться ключевым словом `this` (например, `this.doSomething(5)`). В других языках программирования применять это слово в коде класса необязательно, но компилятор TypeScript сообщит, что не может найти метод без явного использования `this`.

Добавим к классу `Person` открытые сеттер и геттер, чтобы появилась возможность устанавливать и получать значение свойства `_ssn` (листинг Б.9).

Листинг Б.9. Добавление сеттера и геттера

```
class Person {

    constructor(public firstName: string,
        public lastName: string, public age: number, private _ssn?: string) {
    }

    get ssn(): string{
        return this._ssn;
    }

    set ssn(value: string){
```

В этой версии последний аргумент конструктора делается необязательным (_ssn?)

Геттер.

Сеттер.

```
        this._ssn = value;
    }
}

var p = new Person("John", "Smith", 29);
p.ssn = "456-70-1234";

console.log("Last name: " + p.lastName + " SSN: " + p.ssn);
```

Присваивание значения свойству _ssn после создания экземпляра объекта Person с помощью сеттера ssn

В листинге В.9 в геттере и сеттере не содержится никакой логики приложения, а вот в настоящих приложениях в этих методах будет проводиться проверка на приемлемость. Например, код в геттере и сеттере может проверять авторизацию того, кто их вызывает, без которой получить или установить значение `_ssn` невозможно.

ПРИМЕЧАНИЕ

Начиная со спецификации ES5, в JavaScript также поддерживаются геттеры и сеттеры.

Обратите внимание: в методах для доступа к свойству объекта используется ключевое слово `this`. В TypeScript это обязательное условие.

Б.6.3. Наследование

В JavaScript поддерживается прототипное наследование *на основе объектов*, где один объект может использовать другой в качестве прототипа. Для наследования в классах в TypeScript, как в ES6 и в других объектно-ориентированных языках, имеется ключевое слово `extends`. Но в ходе транспиляции в JavaScript в создаваемом коде применяется синтаксис прототипного наследования.

На рис. Б.6 представлено создание класса `Employee` (строка 9), расширяющего класс `Person` (показано на снимке экрана из интерактивной среды TypeScript). В правой части можно увидеть транспилированную JavaScript-версию, в которой используется прототипное наследование. TypeScript-версия кода гораздо лаконичнее и понятнее.

Добавим к классу `Employee` конструктор и свойство `department` (листинг Б.10).

Листинг Б.10. Использование наследования

```
classEmployeeextendsPerson{
    department: string;
    constructor(firstName: string, lastName: string,
            age: number, _ssn: string, department: string){

        super(firstName, lastName, age, _ssn);

        this.department = department;
    }
}
```

Объявление свойства department

Создание конструктора, имеющего дополнительный аргумент department

Подкласс, объявляющий конструктор, должен вызывать конструктор родительского класса

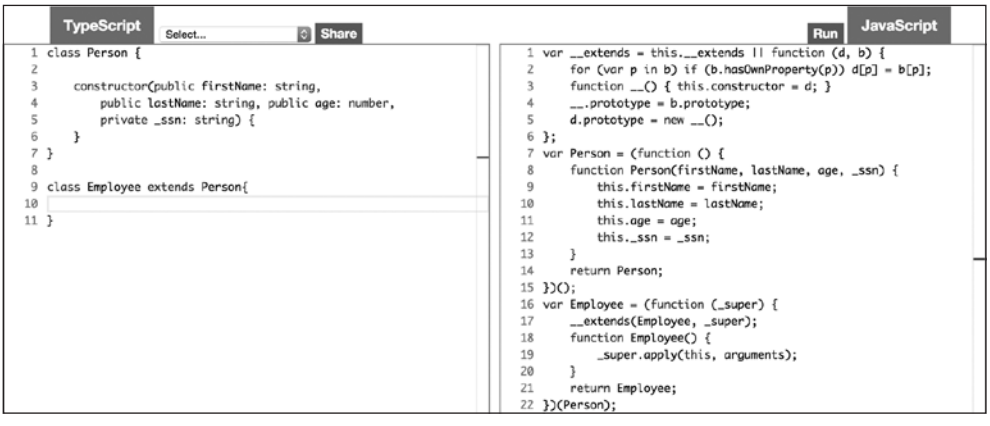


Рис. Б.6. Наследование классов в TypeScript

Если вызвать метод, объявленный в родительском классе, в отношении объекта, относящегося к подклассу, то можно будет воспользоваться именем этого метода, как будто он был объявлен в подклассе. Но иногда необходимо вызвать конкретно тот метод, который определен в родительском классе, и тогда следует задействовать ключевое слово `super`.

Ключевое слово `super` можно применять двумя способами. В конструкторе подкласса оно служит в качестве вызова метода. Кроме того, оно может, в частности, использоваться для вызова метода родительского класса. Обычно оно применяется с переопределением метода. Например, если и родительский класс, и его потомок имеют метод `doSomething()`, то потомок может задействовать функциональность, запрограммированную в родительском классе, и добавить к имеющимся другие свойства:

```
doSomething(){
    super.doSomething();
    // Сюда добавляются дополнительные функциональные свойства
}
```

Дополнительно о ключевом слове `super` можно прочитать в подразделе А.7.4. Мы уже дошли до половины данного приложения, поэтому сделаем передышку и просмотрим то, что изучено до сих пор.

- Несмотря на то что Angular-приложения можно создавать, используя синтаксис JavaScript, соответствующий спецификации ES5 или ES6, применение TypeScript имеет преимущества на стадии разработки вашего проекта.
- TypeScript позволяет объявлять типы элементарных переменных, а также разрабатывать собственные типы. Транспиляторы удаляют информацию о типах, поэтому ваше приложение может быть развернуто на любом браузере, поддерживающем синтаксис ECMAScript 3, 5 или 6.
- Компилятор TypeScript превращает `.ts`-файлы в их `.js`-аналоги. Его можно запустить в режиме отслеживания, тогда проводимые им преобразования станут инициироваться при любом изменении в любом `.ts`-файле.

- Классы TypeScript придают коду более декларативный вид. Концепция классов и наследования хорошо знакома разработчикам, использующим другие объектно-ориентированные языки.
- Модификаторы доступа помогают управлять доступом к элементам класса в ходе разработки, но не так строги, как их аналоги в языках, подобных Java и C#.

Начиная со следующего раздела, мы продолжим знакомство с дополнительными синтаксическими конструкциями TypeScript, но если вы хотите увидеть совместную работу TypeScript и Angular, то можете просто перейти к изучению раздела Б.9.

Б.7. Обобщения

В TypeScript поддерживаются параметризированные типы, известные также как *обобщения*, которые могут использоваться в различных сценариях. Например, можно создать функцию, способную получать значение любого типа, но в ходе ее вызова из конкретного контекста явно указать конкретный тип.

Возьмем еще один пример: в массиве способны храниться объекты любого типа, но можно указать, какие именно типы объектов (например, экземпляры класса `Person`) разрешено там хранить. Если вы (или кто-нибудь еще) попытаетесь добавить объект другого типа, то компилятор TypeScript выдаст ошибку.

В следующем фрагменте кода (листинг Б.11) объявляется класс `Person`, создаются два его экземпляра, которые сохраняются в массиве `workers`, объявленном с обобщенным типом. Такие типы обозначаются путем помещения их в угловые скобки (например, `<Person>`).

Листинг Б.11. Использование обобщенного типа

```
class Person {
    name: string;
}
class Employee extends Person{
    department: number;
}
class Animal {
    breed: string;
}
var workers: Array<Person> = [];
workers[0] = new Person();
workers[1] = new Employee();
workers[2] = new Animal(); // ошибка в ходе компиляции
```

В этом фрагменте кода объявляются классы `Person` (Люди), `Employee` (Работники) и `Animal` (Животные), а также массив `workers` (рабочие) с обобщенным типом `<Person>`. Таким образом анонсируются планы сохранения только экземпляров класса `Person` или его потомков. При попытке сохранения в том же самом массиве экземпляра класса `Animal` в ходе компиляции будет выдана ошибка.

При работе в организации, где в качестве работников выступают животные (например, полицейские собаки), можно изменить объявление `workers` следующим образом:

```
var workers: Array<any> = [];
```

ПРИМЕЧАНИЕ

В разделе Б.8 будет показан еще один пример использования обобщений. Там мы объявим массив workers array, имеющий тип интерфейса.

Можно ли применять обобщенные типы с любым объектом или функцией? Нет. Создатель объекта или функции должен выдать на это разрешение. Если открыть файл определения типов TypeScript (`lib.d.ts`) на GitHub на https://github.com/Microsoft/TypeScript/blob/1344b14bd54854fd8d8993b625d9aca6368d1835/bin/lib.d.ts и провести поиск по строке `interface Array`, можно будет увидеть объявление `Array`, показанное на рис. Б.7. (Файлы определения типов рассматриваются в разделе Б.10.)

```
1004  ////////////////////////////
1005  /// ECMAScript Array API (specially handled by compiler)
1006  ////////////////////////////
1007
1008  interface Array<T> {
1009      /**
1010        * Gets or sets the length of the array. This is a number one higher than the h
1011        */
1012      length: number;
1013      /**
1014        * Returns a string representation of an array.
1015        */
1016      toString(): string;
1017      toLocaleString(): string;
1018      /**
1019        * Appends new elements to an array, and returns the new length of the array.
1020        * @param items New elements of the Array.
1021        */
1022      push(...items: T[]): number;
1023      /**
1024        * Removes the last element from an array and returns it.
1025        */
1026      pop(): T;
1027      /**
1028        * Combines two or more arrays.
1029        * @param items Additional items to add to the end of array1.
1030        */
```

Рис. Б.7. Фрагмент кода lib.d.ts с описанием API Array

Обозначение `<T>` в строке 1008 свидетельствует, что TypeScript позволяет объявлять параметр типа с `Array`, и компилятор будет проверять предоставление в вашей программе конкретного типа. В листинге Б.11 этот обобщенный параметр `<T>` указывается как `<Person>`. Но поскольку обобщения в ES6 не поддерживаются,

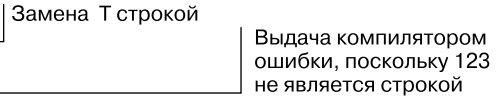
в созданном транспилятором коде вы их не увидите. Это всего лишь дополнительные меры безопасности для разработчиков на этапе компиляции.

В строке 1022 на рис. Б.7 можно увидеть еще один символ `T`. Когда обобщенные типы указываются с аргументами функций, угловые скобки не нужны. Но реальный тип `T` в TypeScript отсутствует. Здесь `T` означает, что метод `push` позволяет внедрять в массив объекты определенного типа, как в следующем примере:

```
workers.push(new Person());
```

В данном разделе проиллюстрирован всего один вариант работы с обобщенными типами в массиве, который уже поддерживает обобщения. Можно также создавать собственные классы или функции, поддерживающие обобщения. Если где-либо в коде будет предпринята попытка вызова функции `saySomething()` с предоставлением неверного типа аргумента, то компилятор TypeScript выдаст ошибку:

```
function saySomething<T>(data: T){

}
saySomething<string>("Hello");
saySomething<string>(123);
```



Созданный код JavaScript не включает никакую информацию, касающуюся обобщения, и предыдущий фрагмент кода будет транспилирован в следующий код:

```
function saySomething(data) {
}
saySomething("Hello");
saySomething(123);
```

Получить более подробную информацию об обобщениях можно в разделе Generics в справочнике по TypeScript (http://www.typescriptlang.org/docs/handbook/generics.html).

Б.8. Интерфейсы

Концепция интерфейсов, которая в других объектно-ориентированных языках используется для введения *кодового контракта*, обязательного к соблюдению со стороны API, в JavaScript не поддерживается. Примером контракта может послужить то, что в классе `X` объявляется реализация интерфейса `Y`. Если класс `X` не будет включать реализацию методов, объявленных в интерфейсе `Y`, то это будет считаться нарушением контракта и код не пройдет компиляцию.

В TypeScript для поддержки интерфейсов включены ключевые слова `interface` и `implements`, но интерфейсы в код JavaScript не транспилируются. Они просто помогают избегать использования неверных типов в ходе разработки.

Для применения интерфейсов в TypeScript имеются две схемы.

- Объявление интерфейса, определяющего пользовательский тип, содержащий несколько свойств. Затем объявление метода, имеющего аргумент такого типа.

При вызове этого метода компилятор проверит, включены ли в объект, переданный в качестве аргумента, все свойства, объявленные в интерфейсе.

- Объявление интерфейса, включающего абстрактные (нереализованные) методы. Когда класс объявляет, что реализует (`implements`) этот интерфейс, данный класс должен предоставить реализацию всех абстрактных методов.

Рассмотрим реализацию этих двух схем на примерах.

Б.8.1. Объявление пользовательских типов с помощью интерфейсов

При использовании JavaScript-сред вам могут попадаться API, требующие в качестве параметра функции некий конфигурационный объект. Чтобы выяснить, какие свойства должны быть предоставлены в этом объекте, нужно либо открыть документацию по API, либо прочитать исходный код среды. В TypeScript можно объявить интерфейс, включающий все свойства и их типы, которые должны присутствовать в объекте конфигурации.

Посмотрим, как это можно сделать в классе `Person`, содержащем конструктор с четырьмя аргументами: `firstName`, `lastName`, `age` и `ssn`. На сей раз будет объявлен интерфейс `IPerson`, содержащий четыре элемента, а конструктор класса `Person` будет изменен для использования в качестве аргумента объекта этого пользовательского типа (листинг Б.12).

Листинг Б.12. Объявление интерфейса

```
interfaceIPerson {

    firstName: string;
    lastName: string;
    age: number;
    ssn?: string;
}

class Person {

    constructor(public config: IPerson) {
    }
}

var aPerson: IPerson = {
    firstName: "John",
    lastName: "Smith",
    age: 29
}

var p = new Person(aPerson);
console.log("Last name: " + p.config.lastName );
```

Объявление интерфейса IPerson с ssn в качестве необязательного элемента (обратите внимание на знак вопроса)

У класса Person имеется конструктор с одним аргументом типа IPerson

Создание литерала объекта aPerson с элементами, совместимыми с IPerson

Создание экземпляра объекта Person, предоставляющего в качестве аргумента объект типа IPerson

В TypeScript имеется структурная система типов, что означает совместимость двух различных типов в том случае, если оба они включают одни и те же элементы. Наличие одинаковых элементов говорит о наличии у них одинаковых имен и типов. В коде листинга Б.12, даже если не указывать тип переменной `aPerson`, она все равно будет считаться совместимой с `IPerson` и может использоваться при создании экземпляра объекта `Person` в качестве аргумента конструктора.

При изменении имени или типа одного из элементов `IPerson` компилятор TypeScript выдаст ошибку. С другой стороны, при попытке создать экземпляр `Person`, содержащий объект со всеми требуемыми элементами `IPerson` и некоторые другие элементы, красный флажок поднят не будет. В качестве аргумента конструктора `Person` можно использовать следующий объект:

```
var anEmployee: IPerson = {
    firstName: "John",
    lastName: "Smith",
    age: 29,
    department: "HR"
}
```

Элемент `department` не был определен в интерфейсе `IPerson`, но, поскольку у объекта имеются все остальные элементы, перечисленные в интерфейсе, условия контракта соблюдены.

В интерфейсе `IPerson` не определяются никакие методы, но в интерфейсы TypeScript могут включаться сигнатуры методов без реализации.

Б.8.2. Использование ключевого слова implements

Ключевое слово `implements` может применяться с объявлением класса, чтобы аннотировать факт реализации классом конкретного интерфейса. Предположим, имеется интерфейс `IPayable`, который объявлялся следующим образом:

```
interface IPayable{
  increase_cap:number;
  increasePay(percent: number): boolean
}
```

Теперь класс `Employee` может объявить, что реализует интерфейс `IPayable`:

```
class Employee implements IPayable{
    // Сюда помещается код реализации
}
```

Прежде чем перейти к подробностям реализации, ответим на следующий вопрос: «Почему бы просто не написать весь требуемый код в классе, не выделяя часть кода в интерфейс?» Предположим, нужно создать приложение, позволяющее поднимать заработную плату работникам вашей организации. Можно создать класс `Employee` (расширяющий класс `Person`) и включить в него метод `increaseSalary()`. Тогда бизнес-аналитики могут попросить вас добавить возмож-

ность повысить выплаты подрядчикам, работающим на вашу фирму. Но подрядчики представлены именами своих компаний и идентификационными номерами, на них не распространяется понятие заработной платы, и расчеты с ними ведутся на почасовой основе.

Вы можете создать еще один класс для подрядчиков по имени `Contractor` (который не является наследником класса `Person`), включающий некие свойства и метод для поднятия почасовой оплаты `increaseHourlyRate()`. Теперь у вас имеются два различных API: один для поднятия заработной платы работникам, а другой для поднятия почасовой выплаты подрядчикам. Более рациональным решением станет создание общего интерфейса `IPayable` и классов `Employee` и `Contractor`, предоставляющих, как показано далее, *разные реализации* `IPayable` для этих классов (листинг Б.13).

Листинг Б.13. Использование нескольких реализаций интерфейса

```
interface IPayable{

  increasePay(percent: number): boolean
}

classPerson {
    // Свойства для краткости опущены

    constructor() {
    }
}

class Employee extends Person implements IPayable{

    increasePay(percent: number): boolean{

        console.log("Increasing salary by "
        ➥             + percent)
        return true;
    }
}

class Contractor implements IPayable{

    increaseCap:number = 20;

    increasePay(percent: number): boolean{
        if (percent < this.increaseCap) {
            console.log("Increasing hourly
            ➥           rate by " + percent)
            return true;
        } else {
            console.log("Sorry, the increase
            ➥           cap for contractors is",
            ➥ this.increaseCap);
```

Интерфейс Ipayable включает сигнатуру метода increasePay(), который будет реализован классами Employee и Contractor

КлассPerson служит в качестве базового для класса Employee

Класс Employee является наследником класса Person и реализует интерфейс IPayable. В классе может быть реализация сразу нескольких интерфейсов

В классе Employee реализуется метод increasePay(). Зарплата работника может быть поднята на любую сумму, поэтому метод просто выдает на консоль сообщение и возвращает значение true (разрешающее выполнить повышение)

Класс Contractor включает свойство, накладывающее ограничение 20 % на повышение выплаты

Реализация increasePay() в классе Contractor иная. Вызов increasePay() с аргументом, превышающим значение 20, приведет к выдаче сообщения Sorry и возвращению false

```
            return false;
        }
    }
}

var workers: Array<IPayable> = [];
workers[0] = new Employee();
workers[1] = new Contractor();

workers.forEach(worker => worker.increasePay(30));
```

Теперь можно будет вызвать метод increasePay() в отношении любого объекта массива workers. Заметьте, что круглые скобки с выражением стрелочной функции, имеющей всего один аргумент worker, не используются

Объявление массива с обобщением <IPayable>, позволяющим помещать в него любые объекты типа IPayable

Запуск кода листинга Б.13 выведет на консоль браузера следующую информацию:

```
Increasing salary by 30
Sorry, the increase cap for contractors is 20
```

Зачем объявлять классы с ключевым словом implements

В листинге Б.13 показано структурное подтипирование TypeScript. Если убрать из объявления либо `Employee`, либо `Contractor` фрагмент кода `implements Payable`, то код сохранит работоспособность и компилятор не станет возражать насчет строк кода, добавляющих эти объекты к массиву `workers`. Компилятор обладает достаточной логикой, чтобы понять, что, даже если в классе реализация `IPayable (implements IPayable)` явным образом не объявлена, метод `increasePay()` в нем все равно должным образом реализуется.

Но если убрать фрагмент кода `implements IPayable` и попытаться изменить сигнатуру метода `increasePay()` в любом из классов, то объект такого класса поместить в массив `workers` не удастся, потому что этот объект уже не будет относиться к типу `IPayable`. Кроме того, без ключевого слова `implements` будет нарушена поддержка IDE-среды (например, проведение с ее помощью реструктуризации кода).

Б.8.3. Использование callable-интерфейсов

В TypeScript имеется интересное свойство, известное как вызываемый, или callable-интерфейс, который содержит пустую сигнатуру функции (сигнатуру без имени функции). В следующем примере кода показана пустая сигнатура функции, получающей один параметр типа `number` и возвращающей значение типа `boolean`:

```
(percent: number): boolean;
```

Пустая сигнатура показывает, что интерфейс относится к разряду вызываемых (callable). В листинге Б.14 будет показана другая версия объявления

IPayable, содержащая пустую сигнатуру функции. Для краткости наследование удалено из данного примера. В нем будут объявлены отдельные функции, реализующие правила для повышения выплат работникам и подрядчикам. Эти функции будут передаваться в качестве аргументов и вызываться конструктором класса Person.

Листинг Б.14. Callable-интерфейс с пустой функцией

```
interface IPayable {
    (percent: number): boolean;
}

class Person {
    constructor(private validator: IPayable) {
    }

    increasePay(percent: number): boolean {
        return this.validator(percent);
    }

}

var forEmployees: IPayable = (percent) => {
        console.log("Increasing salary by ", percent);
        return true;
    };

var forContractors: IPayable = (percent) => {
    var increaseCap: number = 20;

    if (percent < increaseCap) {
            console.log("Increasing hourly rate by", percent);
            return true;
        } else {
            console.log("Sorry, the increase cap for contractors is ",
            ➥ increaseCap);
            return false;
        }

}
var workers: Array<Person> = [];
workers[0] = new Person(forEmployees);
workers[1] = new Person(forContractors);
workers.forEach(worker => worker.increasePay(30));
```

Callable-интерфейс, включающий пустую сигнатуру функции

Конструктор класса Person получает в качестве аргумента реализацию callable-интерфейса IPayable

Метод increasePay() вызывает пустую функцию в отношении переданной реализации Ipayable, предоставляя значение повышения выплаты для проверки

Правила повышения зарплаты для работников реализуются с помощью выражения стрелочной функции

Правила для повышения выплат подрядчикам реализуются с использованием выражения стрелочной функции

Создание двух экземпляров объектов Person с передачей им разных правил повышения выплат

Вызов increasePay() в отношении каждого экземпляра, проверяющий надбавку 30 %

Запуск кода листинга Б.14 выведет на консоль браузера следующую информацию:

```
Increasing salary by 30
Sorry, the increase cap for contractors is 20
```

Интерфейсы поддерживают наследование с помощью ключевого слова `extends`. Если в классе реализуется интерфейс `A`, являющийся расширением интерфейса `B`, то в классе должны быть реализованы все элементы из `A` и `B`.

Трактовка классов как интерфейсов

В TypeScript о любом классе можно думать, как об интерфейсе. Если имеются объявления `class A {}` и `class B {}`, то вполне допустимо будет написать `class A implements B {}`. Пример такого синтаксиса можно увидеть в разделе 4.4.

При транспиляции в JavaScript интерфейсы TypeScript не создают какой-либо вывод, и если поместить в отдельный файл только объявление интерфейса (например, в файл `ipayable.ts`) и скомпилировать его с помощью tsc, то будет создан пустой файл `ipayable.js`. При загрузке кода, импортирующего интерфейс из файла (например, из `ipayable.js`), используя SystemJS, будет получена ошибка, поскольку импортировать пустой файл нельзя. Нужно сообщить SystemJS, что `IPayable` следует рассматривать в качестве модуля, и зарегистрировать его в глобальном реестре System. Этот можно сделать в ходе конфигурирования SystemJS с помощью показанной ниже аннотации `meta`:

```
System.config({
  transpiler: 'typescript',
  typescriptOptions: {emitDecoratorMetadata: true},
  packages: {app: {defaultExtension: 'ts'}},
  meta: {
    'app/ipayable.ts': {
      format: 'es6'
    }
}
```

Кроме предоставления способа создания пользовательских типов и минимизации количества ошибок, связанных с несоответствиями типов, механизм интерфейсов существенно упрощает реализацию шаблона проектирования «Внедрение зависимостей», рассмотренного в главе 4.

На этом краткое введение в интерфейсы завершается. Более подробные сведения о них можно найти в разделе Interfaces публикации TypeScript Handbook (http://www.typescriptlang.org/docs/handbook/interfaces.html).

ПРИМЕЧАНИЕ

Удобным средством создания документации программы на основе комментариев вашего кода TypeScript является утилита TypeDoc. Ее можно получить на сайте www.npmjs.com/package/typedoc.

Мы практически завершили наш обзор синтаксиса TypeScript. Пора объединить его с Angular.

Б.9. Добавление метаданных класса с помощью аннотаций

Существует несколько определений понятия *метаданных*. Популярным определением является такое: метаданные — это данные о данных. Мы рассматриваем метаданные в качестве данных, которые дают описание кода. Декораторы TypeScript предоставляют способ добавления метаданных к вашему коду. В частности, чтобы превратить класс TypeScript в компонент Angular, его можно *проаннотировать* с помощью метаданных. Аннотации начинаются со значка `@`.

Чтобы превратить класс TypeScript в UI-компонент Angular, необходимо отдекорировать его, задействуя аннотацию `@Component`. Angular воспользуется внутренним механизмом для синтаксического разбора ваших аннотаций и создаст код, добавляющий запрошенное поведение к классу TypeScript:

```
@Component({
  // Сюда нужно включить селектор (имя) для идентификации компонента
  // в HTML-документе.
  // Предоставьте свойство шаблона фрагменту HTML, чтобы отобразить компонент.
  // Сюда также помещается стилевое оформление компонента.
})
class HelloWorldComponent {
  // Сюда помещается код, реализующий имеющуюся в компоненте
  // логику приложения.
}
```

При использовании аннотаций необходимо наличие обработчика аннотаций, способный провести синтаксический разбор содержимого аннотации и превратить его в код, понятный механизму выполнения кода (движку JavaScript браузера). Обязанности обработчика аннотаций в контексте данной книги выполняет Angular-компилятор ngc.

Для применения аннотаций, поддерживаемых Angular, нужно импортировать их реализацию в код вашего приложения. Например, следует импортировать аннотацию `@Component` из модуля Angular:

```
import { Component } from 'angular2/core';
```

Хотя реализация данных аннотаций выполнена на Angular, может появиться потребность в стандартизированном механизме для создания своих собственных аннотаций. Именно для этого и предназначены декораторы TypeScript. Их нужно воспринимать следующим образом: Angular предлагает свои аннотации, позволяющие декорировать ваш код, а TypeScript дает вам возможность создавать свои собственные аннотации с поддержкой декораторов.

Б.10. Файлы определения типов

На протяжении нескольких лет большое хранилище файлов определения TypeScript под названием *DefinitelyTyped* было единственным источником определения типов TypeScript для нового ECMAScript API и для сотен популярных сред и библиотек, написанных на JavaScript. Данные файлы предназначались для того, чтобы дать компилятору TypeScript сведения о типах, ожидаемых API этих библиотек. Хотя репозиторий http://definitelytyped.org по-прежнему существует, новый репозиторий файлов определения типов расположился на https://www.npmjs.com/, и именно он использовался во всех примерах кода, приводившихся в данной книге.

Суффиксом для любого имени файла определения служит `d.ts`, и файлы определений можно найти в модулях Angular в подкаталогах папки `node_modules/@angular` после запуска команды `npm install` в соответствии с инструкциями, приведенными в главе 2. Все требуемые файлы вида `*.d.ts` связаны с пакетами Angular npm, и в их отдельной установке нет никакой надобности. Присутствие файлов определений в вашем проекте позволит компилятору TypeScript гарантировать, что ваш код использует при вызове Angular API правильные типы.

Например, Angular-приложения запускаются путем вызова метода `bootstrapModule()` с передачей ему в качестве аргумента корневого модуля вашего приложения. Файл `application_ref.d.ts` включает следующие определения для этой функции:

```
bootstrapModule<M>(moduleType: ConcreteType<M>,
compilerOptions?: CompilerOptions | CompilerOptions[]):
Promise<NgModuleRef<M>>;
```

Прочитав эти определения, вы (и компилятор tsc) узнаете, что данная функция может быть вызвана с одним обязательным параметром типа `ConcreteType` и с необязательным массивом ключей компилятора. Если файл `application_ref.d.ts` не был частью вашего проекта, то компилятор TypeScript позволит вызвать функцию `bootstrapModule` с неправильными параметрами типа или вообще без каких-либо параметров, что приведет к возникновению ошибки в ходе выполнения программы. Но файл `application_ref.d.ts` имеется, поэтому TypeScript выдаст ошибку в ходе компиляции с сообщением Supplied parameters do not match any signature of call target. Файлы определения типов также позволяют IDE-средам показывать контекстно зависимую помощь при написании кода, вызывающего функции Angular или присваивающего значения свойствам объектов.

Б.10.1. Установка файлов определения типов

Чтобы установить файлы определения типов TypeScript для библиотеки или среды, написанной на JavaScript, разработчики использовали диспетчеры определения типов — tsd и Typings. Первое средство уже устарело, поскольку всего лишь позволяет получать файлы вида `*.d.ts` из definitelytyped.org. До выхода TypeScript 2.0 мы пользовались средством Typings (https://github.com/typings/typings), которое позволяло привносить определения типов из произвольно взятого хранилища.

С выходом TypeScript 2.0 необходимость применять диспетчеры определения типов для проектов на основе использования npm отпала. Теперь npm-репозиторий https://www.npmjs.com/ включает структуру `@types`, хранящую определения типов для популярных библиотек JavaScript. Здесь упоминаются все библиотеки, фигурирующие на definitelytyped.org.

Допустим, нужно установить файл определения типов для jQuery. Запуск следующей команды приведет к установке определений типов в каталог `node_modules/@types` и к сохранению этой зависимости в файле `package.json` вашего проекта:

```
npm install @types/jquery --save-dev
```

В данной книге при установке определений типов во многих учебных проектах использовались аналогичные команды. Например, в ES6 для массивов был введен метод, но если ваш TypeScript-проект настроен на выдачу в результате компиляции кода, отвечающего спецификации ES5, то ваша IDE-среда выделит метод `find()` красным цветом, поскольку в ES5 он не поддерживается. Установка файла определения типов для ES6-shim устранит красноту в вашей IDE-среде:

```
npm i @types/es6-shim --save-dev
```

А что будет, если tsc не сможет найти файл определения типов?

В период написания книги (TypeScript 2.0) была вероятность, что tsc не найдет файлы определения типов, находящиеся в каталоге `node_modules/@types`. Если вы столкнетесь с этой проблемой, то добавьте требуемые файлы к разделу `types` файла `tsconfig.json`. Вот пример:

```
"compilerOptions": {
      ...
   "types":["es6-shim", "jasmine"],
}
```

Разрешение модулей и тег reference

Если не используются модули CommonJS, то добавьте к вашему коду TypeScript явное указание ссылки на требуемые определения типов:

```
/// <reference types="typings/jquery.d.ts" />
```

Модули CommonJS применяются как ключ tsc, и каждый проект включает в файле `tsconfig.json` следующий ключ:

```
"module": "commonjs"
```

Когда tsc видит инструкцию `import`, ссылающуюся на модуль, он автоматически пытается найти файл `<имя-модуля>d.ts` в каталоге `node_modules`. Если такой файл не будет найден, то он поднимается на один уровень выше и повторяет процесс. Дополнительные сведения об этой процедуре можно найти в разделе Typings for npm Modules в публикации TypeScript Handbook (http://www.typescriptlang.org/docs/handbook/declaration-files/publishing.html). В будущих выпусках tsc аналогичная стратегия будет реализована для разрешения модулей AMD.

Angular включает все требуемые файлы определений, и вам не нужно применять диспетчер определения типов, если только ваше приложение не использует другие сторонние библиотеки JavaScript. В таком случае для получения контекстно зависимой помощи в вашей IDE-среде их файлы определений придется установить вручную.

В своих `d.ts`-файлах Angular задействует синтаксис ES6, и для большинства модулей можно воспользоваться следующим синтаксисом импорта: `import {Component} from 'angular2/core';`. Определения класса `Component` будут найдены. Вы будете импортировать все остальные модули и компоненты Angular.

Б.10.2. Управление стилем кода с помощью TSLINT

TSLint — средство, которым можно воспользоваться в целях обеспечить написание программы в соответствии с указанными правилами и стилями программирования. TSLint можно настроить на проверку того, что код TypeScript в вашем проекте имеет нужные выравнивания и отступы; имена всех интерфейсов начинаются с заглавной буквы `I`; в именах классов применяется верблюжий регистр (CamelCase) и т. д.

Средство TSLint можно установить глобально с помощью следующей команды:

```
npm install tslint -g
```

Чтобы установить узловой модуль TSLint в каталог вашего проекта, запустите такую команду:

```
npm install tslint
```

Правила, которые нужно применить к вашему коду, указываются в файле конфигурации `tslint.json`. Пример файла правил поставляется вместе с TSLint. Этот файл называется `sample.tslint.json` и находится в каталоге `docs`. При необходимости конкретные правила можно включать и выключать.

Подробности использования TSLint приведены на сайте www.npmjs.com/package/tslint. Ваша IDE-среда может поддерживать проверку соблюдения правил с помощью TSLint в исходном состоянии.

IDE-среды

Мы хотели придать содержимому книги независимость от той или иной IDE-среды и не включали сюда инструкции, характерные для конкретных сред. Но есть несколько IDE, поддерживающих TypeScript. Наиболее популярными из них являются WebStorm, Visual Studio Code, Sublime Text и Atom. Все эти IDE-среды и редакторы работают под управлением Windows, Mac OS и Linux. Если разработка ваших TypeScript/Angular-приложений ведется на компьютере под управлением Windows, то можно воспользоваться средой Visual Studio 2015.

Б.11. Обзор процесса разработки TypeScript и Angular

Процесс разработки и развертывания TypeScript/Angular-приложений состоит из нескольких этапов, их необходимо по возможности максимально автоматизировать. Для достижения данной цели существует несколько способов. Ниже представлен примерный список этапов, которые могут выполняться для создания Angular-приложения.

1. Создание каталога для вашего проекта.
2. Создание файла `package.json` с перечислением всех зависимостей вашего приложения, например пакетов Angular, среды тестирования Jasmine и т. д.
3. Установка всех пакетов и библиотек, перечисленных в `package.json`, с использованием команды `npm install`.
4. Написание кода приложения.
5. Загрузка вашего приложения в браузер с помощью загрузчика SystemJS, который не только загружает, но и транспилирует TypeScript в JavaScript в браузере.
6. Минимизация и объединение вашего кода и ресурсов в пакеты с использованием Webpack и его дополнительных модулей.
7. Копирование файлов в каталог распространения с использованием сценариев npm.

В главе 2 объясняется, как приступить к новому Angular-проекту и работать с диспетчером пакетов npm и загрузчиком модулей SystemJS.

ПРИМЕЧАНИЕ

Angular CLI — утилита командной строки, которая может создать исходную структуру вашего проекта, сгенерировать компоненты и сервисы и подготовить сборки. Средство Angular CLI описано в главе 10.

ПРИМЕЧАНИЕ

В этом приложении не был упомянут вопрос обработки ошибок, но, поскольку TypeScript является расширением JavaScript, обработка ошибок выполняется точно так же, как и в JavaScript. Прочитать о различных типах ошибок можно в статье JavaScript Reference в разделе Error на ресурсе Mozilla Developer Network (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error).

Об обложке

Иллюстрация взята из выпущенного в 1805 г. справочника Сильвена Марешаля (Sylvain Maréchal). Этот справочник состоит из четырех частей и посвящен региональным традициям, связанным с одеждой. Книга впервые была издана в Париже в 1788 г., за год до Французской революции. Каждый рисунок был раскрашен вручную. Данная иллюстрация, подписанная Le Tuteur, или «Учитель», — лишь один из рисунков коллекции Марешаля. Их разнообразие говорит о том, что 200 лет назад мировые города и регионы были уникальны и индивидуальны. В те времена по одежде человека можно было определить, из какого он места.

Одежда с тех пор изменилась, и такое богатое в то время разнообразие ее видов исчезло. Сейчас зачастую сложно только по одежде понять, с какого континента прибыл тот или иной человек. Возможно, мы обменяли культурное разнообразие на более насыщенную личную жизнь — определенно более разнообразную и быструю технологическую жизнь.

В издательстве Manning мы подчеркиваем изобретательность, инициативу и занимательность бизнеса, связанного с компьютерами, создавая обложки книг, на которых отражено богатое разнообразие жизни регионов два века назад, и возвращая к жизни рисунки Марешаля.