Angular
для профессионалов

Адам Фримен

ПИТЕР®

Apress®

Pro Angular

Second Edition

Adam Freeman

Адам Фримен

Angular

для профессионалов

Apress®

ПИТЕР®

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

2018

Адам Фримен

Angular для профессионалов

Серия «Для профессионалов»

Перевел с английского *Е. Матвеев*

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*Н. Римицан*
Художественный редактор	*С. Заматевская*
Корректоры	*С. Беляева, Н. Викторова*
Верстка	*Н. Лукьянова*

ББК 32.988.02-018.1
УДК 004.43

Фримен А.

Ф88 Angular для профессионалов. — СПб.: Питер, 2018. — 800 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-0451-2

Выжмите из Angular — ведущего фреймворка для динамических приложений JavaScript — всё. Адам Фримен начинает с описания MVC и его преимуществ, затем показывает, как эффективно использовать Angular, охватывая все этапы, начиная с основ и до самых передовых возможностей, которые кроются в глубинах этого фреймворка.

Каждая тема изложена четко и лаконично, снабжена большим количеством подробностей, которые позволят вам стать действительно эффективными. Наиболее важные фичи даны без излишних подробностей, но содержат всю необходимую информацию, чтобы вы смогли обойти все подводные камни.

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

ISBN 978-1484223062 англ.
ISBN 978-5-4461-0451-2

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: 09.2017. 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 07.09.17. Формат 70×100/16. Бумага офсетная. Усл. п. л. 64,500. Тираж 1200. 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

Посвящается моей милой жене Джеки Грифит.
И Крохе, которая тоже бывает милой, когда постарается

Об авторе

Адам Фримен (Adam Freeman) — опытный профессионал в области IT, занимавший руководящие должности во многих компаниях. До недавнего времени он занимал посты технического директора и главного инженера в одном из крупнейших банков. Сейчас Адам посвящает свое время в основном написанию книг и бегу на длинные дистанции.

О научном редакторе

Фабио Клаудио Феррачиати (Fabio Claudio Ferracchiati) — старший консультант и ведущий аналитик/разработчик с опытом использования технологий Microsoft. Он работает на BluArancio (*www.bluarancio.com*). Фабио является сертифицированным разработчиком программных решений для .NET, сертифицированным разработчиком приложений для .NET, сертифицированным профессионалом Microsoft, плодовитым писателем и научным редактором. За последние 10 лет он написал ряд статей для итальянских и международных журналов и участвовал в написании более 10 книг по различным областям компьютерной тематики.

От издательства

После выхода оригинального издания на английском языке, автор обновил книгу для Angular 4 and angular-cli. Эти изменения затронули ключевые главы и все листинги. Русскоязычным читателям повезло, мы внесли соответствующие изменения в книгу и теперь вы сможете использовать Angular 4 and angular-cli в своих проектах.

Файлы к книге можно скачать по ссылке www.apress.com/gp/book/9781484223062

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

Мы будем рады узнать ваше мнение!

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

1 Подготовка

Angular заимствует некоторые лучшие аспекты разработки на стороне сервера и использует их для расширения возможностей разметки HTML в браузере. Таким образом закладывается основа, которая упрощает и облегчает создание приложений с расширенной функциональностью. Приложения Angular строятся на базе паттерна проектирования MVC («модель — представление — контроллер»), который ориентирован на создание приложений, обладающих следующими характеристиками:

- *Простота расширения*: если вы понимаете основы, вам будет легко разобраться даже в самом сложном приложении Angular, а это означает, что вы сможете легко расширять приложения для поддержки новой полезной функциональности.
- *Удобство сопровождения*: приложения Angular просты в отладке, в них легко исправляются ошибки, а это означает простоту сопровождения кода в долгосрочной перспективе.
- *Удобство тестирования*: в Angular реализована хорошая поддержка модульного и сквозного тестирования. Следовательно, вы сможете находить и устранять дефекты до того, как ваши пользователи столкнутся с ними.
- *Стандартизация*: Angular работает на основе внутренней функциональности браузера, не создавая никаких препятствий для вашей работы. Это позволяет вам создавать веб-приложения, соответствующие стандартам, в которых задействована новейшая функциональность (например, различные API HTML5), популярные инструменты и фреймворки.

Angular — библиотека JavaScript с открытым кодом, финансированием разработки и сопровождения которой занимается компания Google. Она использовалась в ряде крупнейших и сложнейших веб-приложений. В этой книге вы узнаете все, что необходимо знать для использования Angular в ваших собственных проектах.

Что вам необходимо знать?

Чтобы книга принесла пользу, читатель должен быть знаком с основами веб-разработки, понимать, как работает HTML и CSS, а в идеале обладать практическими знаниями JavaScript. Для тех, кто забыл какие-то подробности, я кратко

напомню основные возможности HTML, CSS и JavaScript, используемые в книге, в главах 4, 5 и 6. Впрочем, вы не найдете здесь подробного справочника по элементам HTML и свойствам CSS. В книге, посвященной Angular, попросту не хватит места для полного описания HTML. Если вам требуется полный справочник по HTML и CSS, я рекомендую вам другую свою книгу, «The Definitive Guide to HTML5».

Много ли в книге примеров?

В книге *очень* много примеров. Angular лучше всего изучать на реальных примерах, поэтому я постарался включить в книгу как можно больше кода. Чтобы довести количество примеров до максимума, я воспользовался простой схемой, чтобы не приводить содержимое файлов снова и снова. Когда файл впервые встречается в главе, я привожу его полное содержимое, как в листинге 1.1. В заголовке листинга указывается имя файла и папка, в которой этот файл создается. При внесении изменений в код измененные команды выделяются жирным шрифтом.

Листинг 1.1. Пример документа

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { ProductComponent } from "./component";
import { FormsModule, ReactiveFormsModule  } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";

@NgModule({
    imports: [BrowserModule, FormsModule, ReactiveFormsModule],
    declarations: [ProductComponent, PaAttrDirective],
    bootstrap: [ProductComponent]
})
export class AppModule { }
```

Этот листинг позаимствован из главы 15. Неважно, что он делает; просто запомните, что при первом использовании каждого файла в главе будет приводиться полный листинг, как в листинге 1.1. Во втором и всех последующих примерах я привожу только измененные элементы, то есть *неполный листинг*. Частичные листинги легко узнать, потому что они начинаются и заканчиваются многоточием (. . .), как показано в листинге 1.2.

Листинг 1.2. Неполный листинг

```
...
<table class="table table-sm table-bordered table-striped">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
    <tr *ngFor="let item of getProducts(); let i = index"
        [pa-attr]="getProducts().length < 6 ? 'bg-success' : 'bg-warning'">
        <td>{{i + 1}}</td>
        <td>{{item.name}}</td>
        <td [pa-attr]="item.category == 'Soccer' ? 'bg-info' : null">
```

```
            {{item.category}}
        </td>
        <td [pa-attr]="'bg-info'">{{item.price}}</td>
    </tr>
</table>
...
```

Листинг 1.2 также взят из главы 15. В нем приводится только контент элемента body, и часть команд выделена жирным шрифтом. Так я привлекаю ваше внимание к части примера, которая демонстрирует описываемый прием или возможность. В неполных листингах приводятся только части, изменившиеся по сравнению с полным листингом, приведенным где-то ранее. В некоторых случаях изменения приходится вносить в разных частях одного файла; тогда я просто опускаю некоторые элементы или команды для краткости, как показано в листинге 1.3.

Листинг 1.3. Часть команд опущена для краткости

```
import { ApplicationRef, Component } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";
import { ProductFormGroup } from "./form.model";

@Component({
    selector: "app",
    templateUrl: "app/template.html"
})
export class ProductComponent {
    model: Model = new Model();
    form: ProductFormGroup = new ProductFormGroup();

    // ...Другие свойства и методы опущены для краткости...

    showTable: boolean = true;
}
```

Эта схема позволила мне включить в книгу больше примеров, однако она усложняет поиск описания конкретных возможностей. По этой причине все главы, в которых описываются возможности Angular, начинаются со сводной таблицы с описанием различных возможностей, представленных в этой главе, и листингов, демонстрирующих их использование.

Где взять примеры кода?

Проекты примеров всех глав этой книги можно загрузить на сайте *www.apress.com*. Архив загружается бесплатно и включает все поддерживающие ресурсы, необходимые для воссоздания примеров без необходимости вводить весь код заново. Загружать код не обязательно, но возможность простого копирования кода в проекты позволит вам легко экспериментировать с примерами.

Как подготовить среду разработки?

В главе 2 вы познакомитесь с Angular на примере создания простого приложения. В ходе рассмотрения примера я объясню, как подготовить среду разработки для работы с Angular.

Как связаться с автором

Если у вас возникнут проблемы с запуском примеров или вы обнаружите ошибки в книге, свяжитесь со мной по адресу *adam@adam-freeman.com*; я постараюсь вам помочь.

Итоги

В этой главе я кратко обрисовал содержимое и структуру книги. Разработку приложений Angular лучше всего изучать на примерах, поэтому в следующей главе мы с ходу возьмемся за дело. Вы узнаете, как подготовить среду разработки и как использовать ее для создания вашего первого приложения Angular.

2 Первое приложение

Лучший способ начать работу с Angular — просто взяться за дело и создать веб-приложение. В этой главе я покажу, как настроить среду разработки, и проведу вас по основным этапам процесса создания простейшего приложения, начиная с построения статического макета функциональности и применения возможностей Angular для построения динамического веб-приложения (пусть и несложного). В главах 7–10 будет рассмотрено создание более сложных и реалистичных приложений Angular, но пока будет достаточно простого примера, который представит основные компоненты приложений Angular и заложит основу для других глав в этой части книги.

Не беспокойтесь, если что-то из материала этой главы покажется непонятным. Поначалу изучение Angular создает немало трудностей, поэтому эта глава всего лишь дает общее представление об основной последовательности разработки приложений Angular и о том, как стыкуются различные компоненты. На первых порах что-то может остаться непонятным, но когда вы перевернете последнюю страницу, вы будете понимать все, что происходит в этой главе, а также многое другое.

Подготовка среды разработки

Разработка Angular-приложений требует определенной подготовки. Далее я объясню, как выполнить необходимые действия для создания вашего первого проекта. Angular широко поддерживается популярными инструментами разработки, так что вы можете выбрать тот вариант, который лучше подходит лично вам.

Установка Node.js

Многие инструменты, используемые для разработки приложений Angular, зависят от Node.js (также известной как Node) — простой и эффективной исполнительной среды для серверных приложений, написанных на JavaScript, которая была создана в 2009 году. Node.js использует ядро JavaScript, задействованное в браузере Chrome, и предоставляет API для выполнения кода JavaScript за пределами окружения браузера.

Среда Node.js пользовалась большим успехом как сервер приложений, но для этой книги она интересна прежде всего тем, что заложила основу для нового поколения кроссплатформенной разработки и средств построения. Некоторые умные решения, принятые командой разработки Node.js, а также кроссплатформенная поддержка со стороны исполнительной среды JavaScript для Chrome открыли замечательные возможности, которые были моментально замечены увлеченными разработчиками инструментальных средств. Короче говоря, среда Node.js стала важнейшим инструментом разработки веб-приложений.

Очень важно, чтобы вы загрузили ту же версию Node.js, которая используется в этой книге. Хотя среда Node.js относительно стабильна, время от времени в API происходят критические изменения, которые могут нарушить работоспособность приведенных в книге примеров.

Я использую версию 6.10.1, самую актуальную версию с долгосрочной поддержкой на момент написания книги. Возможно, вы выберете для своих проектов более свежий выпуск, но для примеров книги следует придерживаться версии 6.10.1. Полный набор всех выпусков 6.10.1 с программами установки для Windows и Mac OS и двоичными пакетами для других платформ доступен по адресу *https://nodejs.org/dist/v6.10.1*.

В ходе установки Node.js не забудьте указать ключ для добавления пути к исполняемым файлам Node.js в переменную окружения. После завершения установки выполните следующую команду:

```
node -v
```

Если установка прошла так, как положено, команда выводит следующий номер версии:

```
V6.10.1
```

Установка Node.js включает менеджер пакетов проекта NPM (Node Package Manager). Выполните следующую команду, чтобы убедиться в том, что NPM работает:

```
npm -v
```

Если все работает так, как должно, выводится следующий номер версии:

```
3.10.10
```

Установка пакета angular-cli

Пакет `angular-cli` стал стандартным инструментом создания и управления пакетами Angular в ходе разработки. В исходной версии книги я показывал, как создавать пакеты Angular «с нуля»; это довольно долгий и ненадежный процесс, который упрощается благодаря `angular-cli`. Чтобы установить `angular-cli`, откройте новую командную строку и введите следующую команду:

```
npm install --global @angular/cli@1.0.0
```

В системе Linux или macOS вам, возможно, придется использовать команду `sudo`.

Установка Git

Система управления версиями Git нужна для управления некоторыми пакетами, необходимыми для разработки Angular. Если вы работаете в Windows или macOS, загрузите и запустите программу установки по адресу *https://git-scm.com/downloads*. (Возможно, в macOS вам придется изменить настройки безопасности для открытия программы установки, которую разработчики не снабдили цифровой подписью.)

Система Git заранее устанавливается в большинстве дистрибутивов Linux. Если вы захотите установить самую свежую версию, обратитесь к инструкциям для своего дистрибутива по адресу *https://git-scm.com/download/linux*. Например, для Ubuntu — моего дистрибутива Linux — используется следующая команда:

```
sudo apt-get install git
```

Завершив установку, откройте новую командную строку и выполните следующую команду, чтобы проверить правильность установки Git:

```
git --version
```

Команда выводит версию установленного пакета Git. На момент написания книги новейшей версией Git для Windows была версия 2.12.0, для macOS — 2.10.1, а для Linux – 2.7.4.

Установка редактора

В разработке приложений Angular может использоваться любой редактор для программистов. Выбор возможных вариантов огромен. В некоторых редакторах предусмотрена расширенная поддержка работы с Angular, включая выделение ключевых терминов и хорошую интеграцию с инструментарием. Если у вас еще нет любимого редактора для разработки веб-приложений, в табл. 2.1 представлены некоторые популярные варианты. Материал книги не зависит от какого-либо конкретного редактора; используйте тот редактор, в котором вам удобнее работать.

Один из важнейших критериев при выборе редактора — возможность фильтрации содержимого проекта, чтобы вы могли сосредоточиться на работе с некоторым подмножеством файлов. Проект Angular может содержать кучу файлов, многие из которых имеют похожие имена, поэтому возможность быстро найти и отредактировать нужный файл чрезвычайно важна. В редакторах эта функция может быть реализована разными способами: с выводом списка файлов, открытых для редактирования, или возможностью исключения файлов с заданным расширением.

ПРИМЕЧАНИЕ

Если вы работаете в Visual Studio (полноценной среде Visual Studio, не в Visual Studio Code), процесс работы с проектами Angular становится еще сложнее, особенно если вы захотите добавить Angular в проект ASP.NET Core MVC. Я планирую выпустить отдельное дополнение по использованию Angular в Visual Studio, вы сможете бесплатно загрузить его из репозитория GitHub этой книги.

Таблица 2.1. Популярные редакторы с поддержкой Angular

Название	Описание
Sublime Text	Sublime Text — коммерческий кроссплатформенный редактор с пакетами для поддержки большинства языков программирования, фреймворков и платформ. За подробностями обращайтесь по адресу www.sublimetext.com
Atom	Atom — бесплатный кроссплатформенный редактор с открытым кодом, уделяющий особое внимание возможностям настройки и расширения. За подробностями обращайтесь по адресу atom.io
Brackets	Brackets — бесплатный редактор с открытым кодом, разработанный компанией Adobe. За подробностями обращайтесь по адресу brackets.io
WebStorm	WebStorm — платный кроссплатформенный редактор с множеством интегрированных инструментов, чтобы вам не пришлось пользоваться командной строкой во время разработки. За подробностями обращайтесь по адресу www.jetbrains.com/webstorm
Visual Studio Code	Visual Studio Code — бесплатный кроссплатформенный редактор с открытым кодом, разработанный компанией Microsoft, с хорошими возможностями расширения. За подробностями обращайтесь по адресу code.visualstudio.com

Установка браузера

Остается принять последнее решение: выбрать браузер, который будет использоваться для проверки результатов работы в ходе разработки. Все браузеры последнего поколения обладают хорошей поддержкой для разработчика и хорошо сочетаются с Angular. Я использовал в книге Google Chrome; этот же браузер я могу порекомендовать и вам.

Создание и подготовка проекта

После того как вы установите Node.js, `angular-cli`, редактор и браузер, в вашем распоряжении окажется все необходимое для запуска процесса разработки.

Создание проекта

Выберите подходящую папку и выполните следующую команду в режиме командной строки, чтобы создать новый проект с именем todo:

```
ng new todo
```

Команда `ng` предоставляется пакетом `angular-cli`, а подкоманда `ng new` создает новый проект. Процесс установки создает папку с именем todo, которая содержит все файлы конфигурации, необходимые для разработки Angular, некоторые временные файлы, упрощающие начальную стадию разработки, и пакеты NPM, необходимые для разработки, запуска и развертывания приложений Angular. (Пакетов NPM довольно много; это означает, что создание проекта может занять много времени.)

Создание файла пакета

NPM использует файл с именем package.json для чтения списка программных пакетов, необходимых для проекта. Файл package.json создается `angular-cli` как часть инфраструктуры проекта, но он содержит только базовые пакеты, необходимые для разработки Angular. Для приложения этой главы понадобится пакет Bootstrap, не входящий в базовый набор пакетов. Отредактируйте файл project.json в папке todo и добавьте пакет Bootstrap, как показано в листинге 2.1.

ВНИМАНИЕ

К тому моменту, когда вы будете читать книгу, будут выпущены новые версии по крайней мере части пакетов из листинга 2.1. Чтобы ваши результаты соответствовали результатам примеров в этой и других главах, очень важно использовать конкретные версии, указанные в листинге. Если у вас возникнут проблемы с примерами этой или какой-либо из последующих глав, попробуйте использовать исходный код из архива, прилагаемого к книге; его можно загрузить на сайте издательства apress.com. Если же положение окажется совсем безвыходным, отправьте мне сообщение по адресу adam@adam-freeman.com, я постараюсь помочь вам.

Листинг 2.1. Содержимое файла package.json в папке todo

```
{
  "name": "todo",
  "version": "0.0.0",
  "license": "MIT",
  "scripts": {
    "ng": "ng",
    "start": "ng serve",
    "build": "ng build",
    "test": "ng test",
    "lint": "ng lint",
    "e2e": "ng e2e"
  },
  "private": true,

  "dependencies": {
    "@angular/common": "^4.0.0",
    "@angular/compiler": "^4.0.0",
    "@angular/core": "^4.0.0",
    "@angular/forms": "^4.0.0",
    "@angular/http": "^4.0.0",
    "@angular/platform-browser": "^4.0.0",
    "@angular/platform-browser-dynamic": "^4.0.0",
    "@angular/router": "^4.0.0",
    "core-js": "^2.4.1",
    "rxjs": "^5.1.0",
    "zone.js": "^0.8.4",
    "bootstrap": "4.0.0-alpha.4"
  },

  "devDependencies": {
    "@angular/cli": "1.0.0",
```

```
    "@angular/compiler-cli": "^4.0.0",
    "@types/jasmine": "2.5.38",
    "@types/node": "~6.0.60",
    "codelyzer": "~2.0.0",
    "jasmine-core": "~2.5.2",
    "jasmine-spec-reporter": "~3.2.0",
    "karma": "~1.4.1",
    "karma-chrome-launcher": "~2.0.0",
    "karma-cli": "~1.0.1",
    "karma-jasmine": "~1.1.0",
    "karma-jasmine-html-reporter": "^0.2.2",
    "karma-coverage-istanbul-reporter": "^0.2.0",
    "protractor": "~5.1.0",
    "ts-node": "~2.0.0",
    "tslint": "~4.5.0",
    "typescript": "~2.2.0"
  }
}
```

В файле package.json перечислены пакеты, необходимые для начала разработки приложений Angular, и некоторые команды для их использования. Все параметры конфигурации будут описаны в главе 11, а пока достаточно понимать, для чего нужна каждая секция файла package.json (табл. 2.2).

Таблица 2.2. Секции файла package.json

Имя	Описание
scripts	Список сценариев, запускаемых в режиме командной строки. Секция scripts в листинге запускает команды, используемые для компиляции исходного кода, и сервер HTTP для разработки
dependencies	Список пакетов NPM, от которых зависит работа веб-приложения. Для каждого пакета указан номер версии. В секции dependencies в листинге перечислены базовые пакеты Angular; библиотеки, от которых зависит Angular; и CSS-библиотека Bootstrap, которая используется для стилевого оформления контента HTML в книге
devDependencies	Список пакетов NPM, которые используются в разработке, но не нужны для работы приложения после его развертывания. В этой секции перечислены пакеты, компилирующие файлы TypeScript, предоставляющие функциональность сервера HTTP в процессе разработки и обеспечивающие тестирование

Установка пакета NPM

Чтобы файл package.json был обработан NPM для загрузки и установки указанного в нем пакета Bootstrap, выполните следующую команду из папки todo:

```
npm install
```

NPM выдает несколько предупреждений относительно обрабатываемых пакетов, но сообщений об ошибках быть не должно.

Запуск сервера

Инструментарий и базовая структура находятся на своих местах; пришло время убедиться в том, что все работает нормально. Выполните следующие команды из папки todo:

```
npm serve --port 3000 --open
```

Команда запускает сервер HTTP для разработки, который был установлен `angular-cli` и настроен для работы с инструментарием разработчика Angular. Запуск занимает немного времени для подготовки проекта, а вывод выглядит примерно так:

```
** NG Live Development Server is running on http://localhost:3000 **
Hash: b8843310528d229c2540
Time: 11251ms
chunk    {0} polyfills.bundle.js, polyfills.bundle.js.map (polyfills) 158 kB {4}
[initial] [rendered]
chunk {1} main.bundle.js, main.bundle.js.map (main) 3.69 kB {3} [initial]
          [rendered]
chunk {2} styles.bundle.js, styles.bundle.js.map (styles) 9.77 kB {4} [initial]
          [rendered]
chunk {3} vendor.bundle.js, vendor.bundle.js.map (vendor) 2.37 MB [initial]
          [rendered]
chunk {4} inline.bundle.js, inline.bundle.js.map (inline) 0 bytes [entry]
          [rendered]
          webpack: Compiled successfully.
```

Не беспокойтесь, если в вашем случае вывод будет выглядеть немного иначе — главное, чтобы после завершения подготовки появилось сообщение `Compiled successfully`. Через несколько секунд откроется окно браузера (рис. 2.1); это означает, что запуск проекта прошел успешно и в нем используется временный контент, сгенерированный `angular-cli`.

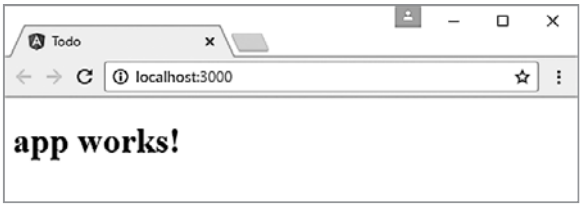


Рис. 2.1. Временный контент HTML

Редактирование файла HTML

Хотя пакет `angular-cli` добавил временный контент, мы сейчас удалим все лишнее и начнем с заготовки HTML, содержащей статический контент. Позже эта заготовка будет расширена для Angular. Отредактируйте файл index.html в папке todo/src и включите в него контент из листинга 2.2.

Листинг 2.2. Содержимое файла index.html в папке todo/src

```
<!DOCTYPE html>
<html>
<head>
    <title>ToDo</title>
    <meta charset="utf-8" />
    <link href="node_modules/bootstrap/dist/css/bootstrap.min.css"
          rel="stylesheet" />
</head>
<body class="m-a-1">

    <h3 class="bg-primary p-a-1">Adam's To Do List</h3>

    <div class="m-t-1 m-b-1">
        <input class="form-control" />
        <button class="btn btn-primary m-t-1">Add</button>
    </div>

    <table class="table table-striped table-bordered">
        <thead>
            <tr>
                <th>Description</th>
                <th>Done</th>
            </tr>
        </thead>
        <tbody>
            <tr><td>Buy Flowers</td><td>No</td></tr>
            <tr><td>Get Shoes</td><td>No</td></tr>
            <tr><td>Collect Tickets</td><td>Yes</td></tr>
            <tr><td>Call Joe</td><td>No</td></tr>
        </tbody>
    </table>
</body>
</html>
```

Сервер HTTP для разработки `angular-cli` добавляет фрагмент JavaScript в контент HTML, передаваемый браузеру. JavaScript открывает обратное подключение к серверу и ждет сигнала на перезагрузку страницы; сигнал отправляется при обнаружении сервером изменений в любых файлах из каталога todo. Как только вы сохраните файл index.html, сервер обнаружит изменения и отправит сигнал. Браузер перезагружается и выводит новый контент (рис. 2.2).

ПРИМЕЧАНИЕ

При внесении изменений в группу файлов может случиться так, что браузер не сможет загрузить и выполнить приложение, особенно в последующих главах, когда приложения станут более сложными. В большинстве случаев сервер HTTP для разработки инициирует перезагрузку в браузере, и все будет нормально, но если у него возникнут затруднения, просто нажмите кнопку обновления в браузере или перейдите по адресу http://localhost:3000, чтобы восстановить нормальную работу.

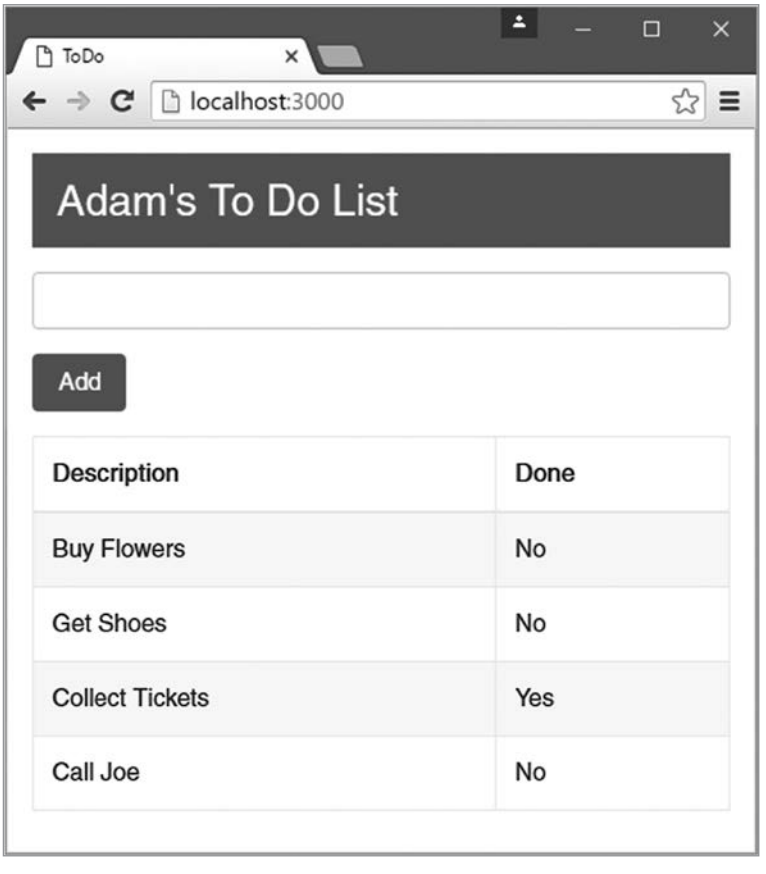


Рис. 2.2. Изменение содержимого файла HTML

Элементы HTML в файле index.html показывают, как будет выглядеть простое приложение Angular, которое мы создадим в этой главе. Его ключевые элементы — заголовок с именем пользователя, поле ввода, кнопка для добавления новой задачи в список и таблица со всеми задачами и признаками их завершения.

В этой книге я использую для оформления контента HTML превосходный CSS-фреймворк Bootstrap. Работа Bootstrap базируется на назначении элементам классов:

```
...
<h3 class="bg-primary p-a-1">Adam's To Do List</h3>
...
```

Элементу `h3` назначены два класса. Класс `bg-primary` назначает цветом фона элемента первичный цвет текущей темы Bootstrap. Я использую тему по умолчанию, в которой первичным цветом является темно-синий; также доступны другие цвета из темы, включая `bg-secondary`, `bg-info` и `bg-danger`. Класс `p-a-1` добавляет ко

всем сторонам элемента фиксированные отступы, чтобы текст был окружен небольшим свободным пространством.

В следующем разделе мы удалим из файла разметку HTML, разобьем ее на несколько меньших фрагментов и используем для создания простого приложения Angular.

> **ИСПОЛЬЗОВАНИЕ ПРЕДВАРИТЕЛЬНОЙ ВЕРСИИ BOOTSTRAP**
>
> В этой книге используется предварительная версия фреймворка Bootstrap. В то время, когда я пишу эти строки, команда Bootstrap занимается разработкой Bootstrap версии 4 и уже опубликовала несколько ранних выпусков. Эти выпуски считаются «альфа-версиями», но они содержат качественный код и работают достаточно стабильно для использования в примерах книги.
>
> Когда мне пришлось выбирать между версией Bootstrap 3, которая скоро станет устаревшей, и предварительной версией Bootstrap 4, я решил использовать новую версию, несмотря на то что некоторые имена классов, используемых для оформления элементов HTML, могут измениться в окончательной версии. Это означает, что для получения предполагаемых результатов в примерах вы должны использовать ту же версию Bootstrap — как и в остальных пакетах, перечисленных в файле package.json из листинга 2.1.

Добавление функциональности Angular в проект

Статическая разметка HTML в файле index.html заменяет простейшее приложение. Пользователь может просматривать список задач, помечать выполненные и создавать новые задачи. В дальнейших разделах мы добавим в проект Angular и воспользуемся некоторыми базовыми возможностями для того, чтобы вдохнуть жизнь в приложение. Для простоты предполагается, что у приложения всего один пользователь и нам не нужно беспокоиться о сохранении данных в приложении, а это означает, что изменения в списке будут потеряны при закрытии или перезагрузке окна браузера. (В последующих примерах, включая приложение SportsStore, создаваемое в главах 7–10, будет продемонстрирован механизм долгосрочного хранения данных.)

Подготовка файла HTML

Первым шагом на пути включения Angular в приложение станет подготовка файла index.html (листинг 2.3).

Листинг 2.3. Подготовка к использованию Angular в файле index.html

```
<!DOCTYPE html>
<html>
<head>
    <title>ToDo</title>
```

```
    <meta charset="utf-8" />
    <link href="node_modules/bootstrap/dist/css/bootstrap.min.css"
          rel="stylesheet" />
</head>
<body class="m-a-1">
    <todo-app>Angular placeholder</todo-app>
</body>
</html>
```

В листинге 2.3 содержимое элемента `body` заменяется элементом `todo-app`. В спецификации HTML элемент `todo-app` отсутствует, и браузер игнорирует его при разборе файла HTML, но этот элемент станет отправной точкой в мире Angular — он заменяется контентом приложения. Если сохранить файл index.html, браузер перезагрузит файл и выведет временный текст (рис. 2.3).

ПРИМЕЧАНИЕ

Если вы воспроизводили примеры из первоначальной версии книги, возможно, вас интересует, почему я не добавил в файл HTML элементы script для встраивания функциональности Angular? Проект, созданный на базе angular-cli, использует инструмент Web Pack, который автоматически генерирует файлы JavaScript для проекта и автоматически встраивает их в файлы HTML, отправляемые браузеру сервером HTTP для разработки.

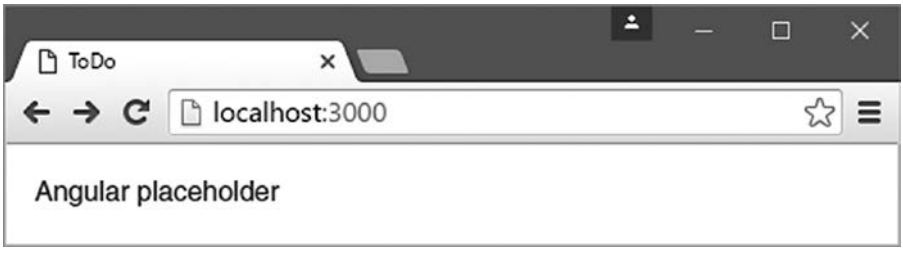


Рис. 2.3. Подготовка файла HTML

Создание модели данных

Когда я создавал статическую заготовку приложения, данные были распределены между всеми элементами HTML.

Имя пользователя содержалось в заголовке:

```
...
<h3 class="bg-primary p-a-1">Adam's To Do List</h3>
...
```

Описания задач содержатся в элементах `td` таблицы:

```
...
<tr><td>Buy Flowers</td><td>No</td></tr>
...
```

Следующая задача — объединить все данные для создания модели данных. Отделение данных от способа их представления для пользователя — один из ключевых принципов паттерна MVC (об этом я расскажу в главе 3).

ПРИМЕЧАНИЕ

Здесь я немного упрощаю. Модель также может содержать логику создания, загрузки, сохранения и изменения объектов данных. В приложениях Angular эта логика часто располагается на сервере, где к ней обращаются веб-службы. За дополнительной информацией обращайтесь к главе 3, а примеры приведены в главе 24.

Приложения Angular обычно пишутся на TypeScript. В главе 6 я расскажу о языке TypeScript, объясню, как он работает и для чего он нужен. TypeScript является надмножеством JavaScript, но одно из его главных преимуществ заключается в том, что он позволяет писать код с использованием новейшей спецификации языка JavaScript, часть возможностей которого не поддерживается некоторыми браузерами, способными к выполнению приложений Angular. Среди прочих пакетов, добавленных `angular-cli` в проект в предыдущем разделе, был компилятор TypeScript, который я настроил для автоматического генерирования совместимых файлов JavaScript при обнаружении изменений в файлах TypeScript.

Чтобы создать модель данных приложения, я добавил файл с именем model.ts в папку todo/src/app (файлы TypeScript имеют расширение .ts) и включил код из листинга 2.4.

Листинг 2.4. Содержимое файла model.ts в папке todo/app

```
var model = {
    user: "Adam",
    items: [{ action: "Buy Flowers", done: false },
    { action: "Get Shoes", done: false },
    { action: "Collect Tickets", done: true },
    { action: "Call Joe", done: false }]
};
```

Одна из важнейших особенностей TypeScript заключается в том, что вы можете писать «нормальный» код JavaScript так, как если бы вы писали его непосредственно для браузера. В листинге я использовал синтаксис объектных литералов JavaScript для присваивания значения глобальной переменной с именем `model`. Объект модели данных содержит свойство `user`, в котором хранится имя пользователя, и свойство `items`, которому присваивается массив объектов со свойствами `action` и `done`; каждый объект представляет задачу в списке.

ПРИМЕЧАНИЕ

В отличие от конфигурации проекта, которую я использовал для более ранних версий Angular, конфигурация, созданная angular-cli, компилирует только те файлы, которые необходимы для передачи приложения браузеру. Это означает, что файл model.ts не будет компилироваться вплоть до его интеграции в функциональность Angular для приложения далее в этой главе.

Этот фрагмент представляет самый важный аспект использования TypeScript: вы не обязаны использовать специфические функции TypeScript и можете писать целые приложения Angular с использованием только возможностей JavaScript, поддерживаемых всеми браузерами (как в листинге 2.4).

Но полезность TypeScript также в значительной мере обусловлена и тем, что TypeScript берет код, использующий новейшие возможности языка JavaScript, в код, который будет работать везде — даже в браузерах, не поддерживающих эту функциональность. В листинге 2.5 приведена модель данных, переписанная для использования новых возможностей JavaScript, добавленных в стандарте ECMAScript 6 (также называемом ES6).

Листинг 2.5. Использование возможностей ES6 в файле model.ts

```
export class Model {
    user;
    items;
    constructor() {
        this.user = "Adam";
        this.items = [new TodoItem("Buy Flowers", false),
                      new TodoItem("Get Shoes", false),
                      new TodoItem("Collect Tickets", false),
                      new TodoItem("Call Joe", false)]
    }
}

export class TodoItem {
    action;
    done;

    constructor(action, done) {
        this.action = action;
        this.done = done;
    }
}
```

Это стандартный код JavaScript, однако ключевое слово `class` было добавлено в поздней версии языка, неизвестной многим разработчикам веб-приложений, и оно не поддерживается старыми браузерами. Ключевое слово `class` используется для определения типов, экземпляры которых создаются ключевым словом `new`; созданные таким образом объекты обладают четко определенными данными и поведением.

Многие возможности, появившиеся в новых версиях JavaScript, представляют собой «синтаксические удобства» для предотвращения наиболее распространенных ловушек JavaScript, таких как непривычная система типов. Ключевое слово `class` не влияет на работу с типами в JavaScript; оно всего лишь делает их более знакомыми и удобными для программистов с опытом работы на других языках (таких, как C# или Java). Мне нравится система типов JavaScript, динамичная и выразительная, но работа с классами более предсказуема и лучше защищена от ошибок. Кроме того, классы упрощают работу с фреймворком Angular, спроектированным с учетом новейших возможностей JavaScript.

ПРИМЕЧАНИЕ

Не огорчайтесь, если какие-то новые возможности, добавленные в новых версиях спецификации JavaScript, вам неизвестны. В главах 5 и 6 рассказано о том, как писать код JavaScript с использованием средств, упрощающих работу с Angular. В главе 6 также описаны некоторые полезные возможности, ориентированные на TypeScript.

Ключевое слово `export` относится к работе с модулями JavaScript. При использовании модулей каждый файл TypeScript или JavaScript считается автономным блоком функциональности, а ключевое слово `export` идентифицирует данные и типы, которые должны использоваться в других местах приложения. Модули JavaScript используются для управления зависимостями между файлами проекта, а также для того, чтобы разработчику не приходилось вручную управлять сложным набором элементов `script` в файле HTML. За подробной информацией о том, как работают модули, обращайтесь к главе 7.

Создание шаблона

Нам понадобится механизм отображения значений данных из модели. В Angular это делается при помощи шаблона — фрагмента HTML с инструкциями, выполняемыми Angular. В ходе создания проекта пакет `angular-cli` создал файл шаблона с именем app.component.html в папке todo/src/app. Отредактируйте этот файл и добавьте разметку из листинга 2.6. Имя файла следует стандартным соглашениям об именах Angular, смысл которых я объясню позднее.

Листинг 2.6. Содержимое файла app.component.html в папке todo/app

```
<h3 class="bg-primary p-a-1">{{getName()}}'s To Do List</h3>
```

Вскоре в этот файл будут добавлены другие элементы, а для начала работы хватит одного элемента `h3`. Для включения значения данных в шаблоне используются двойные фигурные скобки `{{` и `}}`; Angular вычисляет заключенное в них выражение и получает значение для вывода.

Символы `{{` и `}}` обозначают *привязку данных*; иначе говоря, они создают отношения между шаблоном и значением данных. Привязка данных принадлежит к числу важных особенностей Angular. Другие примеры ее использования встретятся вам в этой главе, когда мы займемся расширением функциональности примера; более подробное описание будет приведено в дальнейших главах. В данном случае привязка данных приказывает Angular вызвать функциональность с именем `getName` и использовать результат как содержимое элемента `h3`. Функции `getName` в приложении пока нет, но мы создадим ее в следующем разделе.

Создание компонента

Компонент (component) Angular отвечает за управление шаблоном и передачу ему необходимых данных и логики. Если это утверждение покажется вам слишком

громким, именно компоненты выполняют большую часть тяжелой работы в приложениях Angular. Как следствие, они могут использоваться для самых разных целей.

На данный момент у нас имеется модель данных со свойством `user`, содержащим выводимое имя, а также шаблон, который выводит имя вызовом свойства `getName`. Не хватает компонента, который бы заполнял пробел между ними. Пакет `angular-cli` создал файл компонента с именем app.component.ts в папку todo/src/app; замените исходное содержимое этого файла кодом из листинга 2.7.

Листинг 2.7. Содержимое файла app.component.ts в папке todo/src/app

```
import { Component } from "@angular/core";
import { Model } from "./model";

@Component({
    selector: "todo-app",
    templateUrl: "app/app.component.html"
})
export class AppComponent {
    model = new Model();

    getName() {
        return this.model.user;
    }
}
```

Перед вами все тот же код JavaScript, но в нем встречаются возможности, которые вам, может быть, прежде не встречались; тем не менее эти возможности лежат в основе разработки Angular. В коде можно выделить три основные части, описанные ниже.

Импортирование

Ключевое слово `import` составляет пару для ключевого слова `export` и используется для объявления зависимости от содержимого модуля JavaScript. Ключевое слово `import` встречается дважды в листинге 2.8:

```
...
import { Component } from "@angular/core";
import { Model } from "./model";
...
```

Первая команда `import` используется для загрузки модуля `@angular/core`, содержащего ключевую функциональность Angular, в том числе и поддержку компонентов. При работе с модулями команда `import` перечисляет импортируемые типы в фигурных скобках. В данном случае команда `import` используется для загрузки типа `Component` из модуля. Модуль `@angular/core` содержит многочисленные классы, упакованные вместе, чтобы браузер мог загрузить их в одном файле JavaScript.

Вторая команда `import` загружает класс `Model` из файла в проекте. Цель такого рода команд импортирования начинается с `./`; это означает, что местонахождение модуля определяется относительно текущего файла.

Обратите внимание: ни в одной команде `import` не указано расширение файла. Дело в том, что отношениями между целью команды `import` и файлом, загружаемым браузером, управляет *загрузчик модуля*, настройка которого описана в разделе «А теперь все вместе».

Декораторы

Пожалуй, самая странная часть листинга выглядит так:

```
...
@Component({
    selector: "todo-app",
    templateUrl: "app/app.component.html"
})
...
```

Это *декоратор* (decorator), предоставляющий метаданные о классе. Декоратор `@Component`, как подсказывает его имя, сообщает Angular, что это компонент. Декоратор передает конфигурационную информацию в своих свойствах; `@Component` включает свойства с именами `selector` и `templateUrl`.

Свойство `selector` задает селектор CSS, соответствующий элементу HTML, к которому будет применен компонент. В данном случае я указал элемент `todo-app`, добавленный в файл index.html в листинге 2.3. При запуске приложения Angular сканирует разметку HTML текущего документа и ищет элементы, соответствующие компонентам. Angular находит элемент `todo-app` и понимает, что его нужно передать под контроль этому компоненту.

Свойство `templateUrl` сообщает Angular, где найти шаблон компонента, в данном случае это файл app.component.html в папке app компонента. Далее будут описаны другие свойства, которые могут использоваться с `@Component` и другими декораторами, поддерживаемыми Angular.

Класс

В последней части листинга определяется класс, экземпляр которого создается Angular для создания компонента.

```
...
export class AppComponent {
    model = new Model();

    getName() {
        return this.model.user;
    }
}
...
```

Эти команды определяют класс `AppComponent` со свойством `model` и функцией `getName`, которая предоставляет функциональность, необходимую для работы привязки данных в шаблоне из листинга 2.6.

При создании нового экземпляра класса `AppComponent` свойству `model` присваивается новый экземпляр класса `Model` из листинга 2.5. Функция `getName` возвращает значение свойства `user`, определяемого объектом `Model`.

А теперь все вместе

У нас есть все три ключевых блока функциональности, необходимых для построения простого приложения Angular: модель, шаблон и компонент. При сохранении изменений в файле app.component.ts функциональности было достаточно для того, чтобы связать их воедино и вывести результат, показанный на рис. 2.4.

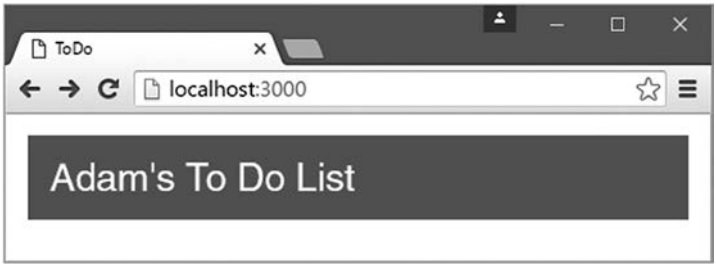


Рис. 2.4. Приложение заработало

Одно из преимуществ использования `angular-cli` при создании проекта заключается в том, что вам не нужно беспокоиться о создании основных файлов, необходимых для приложения Angular. Недостаток — в том, что, пропуская эти файлы, вы можете упустить важные подробности, заслуживающие внимания.

В приложениях Angular должен присутствовать *модуль*. Из-за неудачного выбора термина в разработке приложений Angular встречаются модули двух видов. *Модуль JavaScript* — файл с функциональностью JavaScript, для присоединения которого используется ключевое слово `import`. К другой категории относятся *модули Angular*, используемые для описания приложений или групп взаимосвязанных возможностей. У каждого приложения имеется *корневой модуль*, который предоставляет Angular информацию, необходимую для запуска приложения.

При создании проекта с использованием `angular-cli` был создан файл с именем app.module.ts (стандартное имя файла корневого модуля) в папке todo/src/app. В файл был включен код из листинга 2.8.

Листинг 2.8. Содержимое файла app.module.ts в папке todo/src/app

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpModule } from '@angular/http';
```

```
import { AppComponent } from './app.component';

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

Основная цель модуля Angular — передача конфигурационной информации в свойствах, определяемых декоратором `@NgModule`. Работа модулей подробно объясняется в главе 21, а пока достаточно знать, что свойство `imports` декоратора сообщает Angular, что приложение зависит от функций, необходимых для запуска приложений в браузере, а свойства `declarations` и `bootstrap` сообщают Angular о компонентах приложения и о том, какой компонент должен использоваться для запуска приложения (в этом простом примере компонент только один, поэтому он является значением обоих свойств).

Приложениям Angular также необходим *файл начальной загрузки* с кодом, нужным для запуска приложения и загрузки модуля Angular. Чтобы создать файл начальной загрузки, я создал файл с именем main.ts в папке todo/src и добавил в него код из листинга 2.9.

Листинг 2.9. Содержимое файла main.ts в папке todo/src

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

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) {
  enableProdMode();
}

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

Хотя в этой книге основное внимание уделяется приложениям, работающим в браузере, технология Angular рассчитана на работу в широком диапазоне сред. Команды в файле начальной загрузки выбирают платформу, которая должна использоваться, и загружают корневой модуль, который является точкой входа в приложение.

ПРИМЕЧАНИЕ

Вызов метода platformBrowserDynamic().bootstrapModule предназначен для браузерных приложений — основной темы этой книги. Если вы работаете на разных платформах (скажем, с фреймворком мобильной разработки ionic), используйте другой метод начальной загрузки для той платформы, с которой вы работаете. Разработчики каждой платформы, поддерживающей Angular, предоставляют подробную информацию о методе начальной загрузки для своей платформы.

Браузер выполнил код из файла начальной загрузки; это привело к активизации среды Angular, которая, в свою очередь, обработала документ HTML и обнаружила элемент `todo-app`. Свойство `selector`, использованное для определения компонента, совпадает с элементом `todo-app`, вследствие чего Angular удаляет временный контент и зменяет его шаблоном компонента, автоматически загружаемым из файла app.component.html. В ходе разбора шаблона обнаруживается конструкция привязки `{{ }}`; содержащееся в ней выражение вычисляется, вызывается метод `getName` и выводится результат. Возможно, этот результат не особо впечатляет, но это хорошее начало, и оно закладывает основу для дальнейшего расширения функциональности.

ПРИМЕЧАНИЕ

В любом проекте Angular существует период, в котором вы определяете главные части приложения и связываете их между собой. В это время может появиться впечатление, что вы проделываете значительную работу без особой отдачи. Поверьте, этот период начальных вложений непременно окупится! Более масштабный пример приведен в главе 7, когда мы начнем строить более сложное и реалистичное приложение Angular; оно потребует значительной исходной подготовки и настройки конфигурации, но затем все составляющие быстро встают на свои места.

Расширение функциональности приложения

Итак, основная структура приложения готова, и мы можем заняться добавлением остальных возможностей, которые были смоделированы статической разметкой HTML в начале главы. В следующих разделах будет добавлена таблица со списком задач, а также элемент ввода и кнопка для создания новых записей.

Добавление таблицы

Возможности шаблонов Angular выходят далеко за рамки простого вывода значений данных. Полный спектр возможностей шаблонов будет описан далее, но в приложении-примере мы ограничимся возможностью добавления набора элементов HTML в DOM для каждого объекта в массиве. В данном случае таким массивом будет набор задач из модели данных. Для начала в листинге 2.10 в компонент добавляется метод, который предоставляет шаблону массив задач.

Листинг 2.10. Добавление метода в файл app.component.ts

```
import { Component } from "@angular/core";
import { Model } from "./model";

@Component({
    selector: "todo-app",
    templateUrl: "app/app.component.html"
})
export class AppComponent {
    model = new Model();
```

```
    getName() {
        return this.model.user;
    }

    getTodoItems() {
        return this.model.items;
    }
}
```

Метод `getTodoItems` возвращает значение свойства `items` из объекта `Model`. В листинге 2.11 шаблон компонента обновляется с использованием нового метода.

Листинг 2.11. Вывод списка задач в файле app.component.html

```
<h3 class="bg-primary p-a-1">{{getName()}}'s To Do List</h3>

<table class="table table-striped table-bordered">
    <thead>
        <tr><th></th><th>Description</th><th>Done</th></tr>
    </thead>
    <tbody>
        <tr *ngFor="let item of getTodoItems(); let i = index">
            <td>{{ i + 1 }}</td>
            <td>{{ item.action }}</td>
            <td [ngSwitch]="item.done">
                <span *ngSwitchCase="true">Yes</span>
                <span *ngSwitchDefault>No</span>
            </td>
        </tr>
    </tbody>
</table>
```

Изменения в шаблоне основаны на нескольких разных возможностях Angular. Первая — выражение `*ngFor`, используемое для повторения блока содержимого для каждого элемента массива. Это пример директивы, которые будут рассматриваться в главах 12–16 (директивы — важная часть разработки Angular). Выражение `*ngFor` применяется к атрибуту элемента:

```
...
<tr *ngFor="let item of getTodoItems(); let i = index">
...
```

Это выражение приказывает Angular рассматривать элемент `tr`, к которому оно было применено, как шаблон, который должен быть повторен для каждого объекта, возвращаемого методом `getTodoItems` компонента. Часть `let item` указывает, что каждый объект должен присваиваться переменной с именем `item` для последующих ссылок на него из шаблона.

Выражение `ngFor` также отслеживает индекс текущего обрабатываемого объекта в массиве; он присваивается второй переменной с именем `i`:

```
...
<tr *ngFor="let item of getTodoItems(); let i = index">
...
```

В результате элемент `tr` и его содержимое будут продублированы и вставлены в документ HTML для каждого объекта, возвращаемого методом `getTodoItems`; при каждой итерации к объекту текущей задачи можно обращаться через переменную с именем `item`, а к позиции объекта в массиве — через переменную с именем `i`.

ПРИМЕЧАНИЕ

Не забывайте о символе * при использовании *ngFor. В главе 16 я объясню, что он означает.

В шаблоне `tr` используются две привязки данных, которые можно узнать по символам `{{` и `}}`:

```
...
<td>{{ i + 1 }}</td>
<td>{{ item.action }}</td>
...
```

Эти привязки относятся к переменным, созданным выражением `*ngFor`. Привязки используются не только для ссылок на свойства и имена методов; они также могут использоваться для выполнения простых операций. Пример такого рода встречается в первой привязке, где переменная `i` суммируется с 1.

ПРИМЕЧАНИЕ

Для простых преобразований выражения JavaScript могут встраиваться прямо в привязки, но для более сложных операций в Angular предусмотрен механизм каналов (pipes), который будет описан в главе 18.

Остальные выражения в шаблоне `tr` демонстрируют, как происходит избирательное генерирование контента.

```
...
<td [ngSwitch]="item.done">
    <span *ngSwitchCase="true">Yes</span>
    <span *ngSwitchDefault>No</span>
</td>
...
```

Выражение `[ngSwitch]` представляет собой условную конструкцию, которая используется для вставки в документ разных наборов элементов в зависимости от указанного значения, которым в данном случае является свойство `item.done`. В элемент `td` вложены два элемента `span` с пометкой `*ngSwitchCase` и `*ngSwitchDefault`, которые эквивалентны ключевым словам `case` и `default` обычной конструкции `switch` в языке JavaScript. Я подробно опишу `ngSwitch` в главе 13 (а смысл квадратных скобок — в главе 12), но в результате первый элемент `span` добавляется в документ, когда значение свойства `item.done` истинно, а второй элемент `span` — когда значение `item.done` ложно. В результате значение `true`/`false` свойства `item.done` преобразуется в элементы `span`, содержащие текст `Yes` или `No`. Когда вы сохраняете изменения в шаблоне, браузер перезагружается и таблица задач выводится в браузере (рис. 2.5).

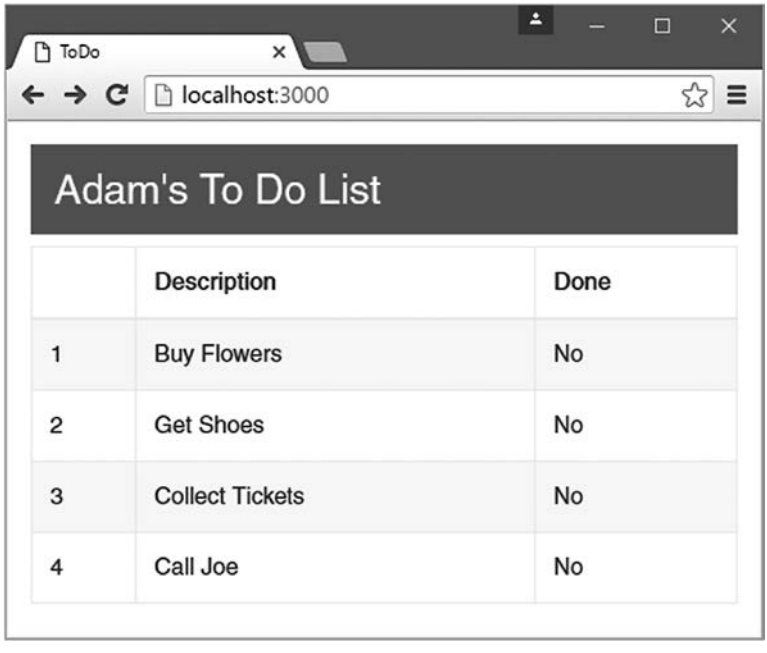


Рис. 2.5. Вывод таблицы задач

Если вы воспользуетесь инструментарием разработчика F12 в браузере, то увидите контент HTML, сгенерированный шаблоном. (Просмотр исходного кода страницы для этого не подойдет — в нем отображается только разметка HTML, отправленная сервером, без учета изменений, вносимых Angular через DOM API.)

Итак, для каждого объекта задачи из модели в таблице создается строка, которая заполняется значениями локальных переменных `item` и `i`, а конструкция `switch` выводит текст Yes или No как признак завершения задачи.

```
...
<tr>
    <td>2</td>
    <td>Get Shoes</td>
    <td><span>No</span></td>
</tr>
<tr>
    <td>3</td>
    <td>Collect Tickets</td>
    <td><span>Yes</span></td>
</tr>
...
```

Создание двусторонней привязки данных

На данный момент шаблон содержит только *односторонние* привязки данных; это означает, что они используются только для вывода значения данных, но ничего не делают для его изменения. Angular также поддерживает двусторонние привяз-

ки данных, которые могут использоваться как для вывода, так и для обновления. Двусторонние привязки используются с элементами форм HTML. Листинг 2.12 добавляет в шаблон флажок, с помощью которого пользователи могут помечать задачи как выполненные.

Листинг 2.12. Добавление двусторонней привязки данных в файл app.component.html

```
<h3 class="bg-primary p-a-1">{{getName()}}'s To Do List</h3>

<table class="table table-striped table-bordered">
    <thead>
        <tr><th></th><th>Description</th><th>Done</th></tr>
    </thead>
    <tbody>
        <tr *ngFor="let item of getTodoItems(); let i = index">
            <td>{{i + 1}}</td>
            <td>{{item.action}}</td>
            <td><input type="checkbox" [(ngModel)]="item.done" /></td>
            <td [ngSwitch]="item.done">
                <span *ngSwitchCase="true">Yes</span>
                <span *ngSwitchDefault>No</span>
            </td>
        </tr>
    </tbody>
</table>
```

Шаблонное выражение `ngModel` создает двустороннюю привязку между значением данных (свойство `item.done` в данном случае) и элементом формы. Сохранив изменения в шаблоне, вы увидите, что в таблице появился новый столбец с флажками. Исходное состояние флажка задается свойством `item.done`, как и при обычной односторонней привязке, но когда пользователь изменяет состояние флажка, Angular реагирует обновлением указанного свойства модели.

Чтобы показать, как работает этот механизм, я оставил столбец с выводом значения свойства `done` в виде текста Yes/No, сгенерированный выражением `ngSwitch` в шаблоне. Когда вы изменяете состояние флажка, соответствующее значение Yes/No тоже изменяется (рис. 2.6).

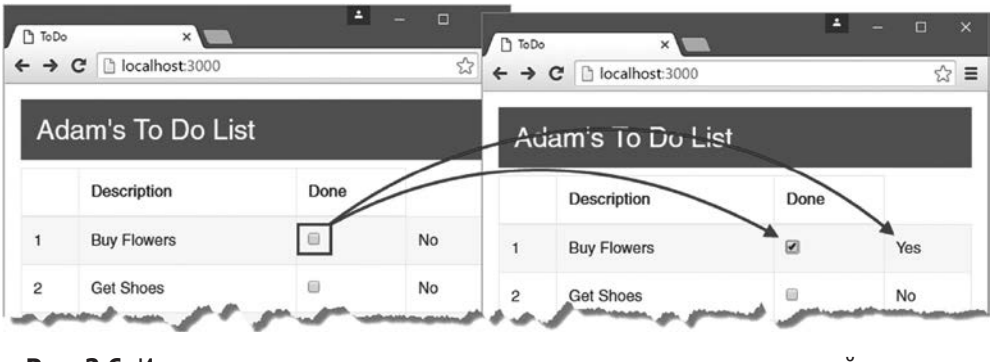


Рис. 2.6. Изменение значения из модели с использованием двусторонней привязки

Здесь проявляется важная особенность Angular: *живая* модель данных. Это означает, что привязки данных — даже односторонние — обновляются при изменении модели данных. Живая модель данных упрощает разработку веб-приложений, потому что вам не придется беспокоиться об обновлении изображения при изменении состояния приложения.

Фильтрация задач

Флажки обеспечивают обновление модели данных. Следующим шагом должно стать исключение задач, которые были помечены как выполненные. В листинге 2.13 метод `getTodoItems` изменяется так, чтобы он отфильтровывал все выполненные задачи.

Листинг 2.13. Фильтрация задач в файле app.component.ts

```
import { Component } from "@angular/core";
import { Model } from "./model";

@Component({
    selector: "todo-app",
    templateUrl: "app.component.html"
})
export class AppComponent {
    model = new Model();

    getName() {
        return this.model.user;
    }

    getTodoItems() {
        return this.model.items.filter(item => !item.done);
    }
}
```

Перед вами пример *лямбда-функции* — более компактного синтаксиса выражения стандартных функций JavaScript. Лямбда-выражения были недавно добавлены в спецификацию языка JavaScript; они предоставляют альтернативу для традиционного использования функций в аргументах, как в следующем примере:

```
...
return this.model.items.filter(function (item) { return !item.done });
...
```

Какой бы способ ни был выбран для определения выражения, передаваемого методу `filter`, в результате будут отображаться только незавершенные задачи. Поскольку модель данных является живой, а изменения отражаются в привязках данных немедленно, при установке флажка задача исключается из представления (рис. 2.7).

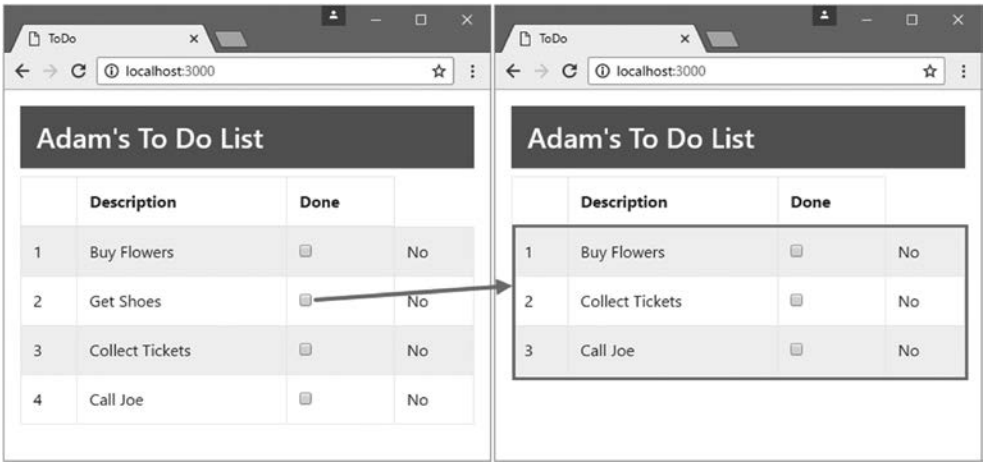


Рис. 2.7. Фильтрация задач

Добавление задач

На следующем шаге мы расширим базовую функциональность, чтобы пользователь мог создавать новые задачи и сохранять их в модели данных. В листинге 2.14 в шаблон компонента добавляются новые элементы.

Листинг 2.14. Добавление элементов в файл app.component.html

```
<h3 class="bg-primary p-a-1">{{getName()}}'s To Do List</h3>
<div class="m-t-1 m-b-1">
    <input class="form-control" #todoText />
    <button class="btn btn-primary m-t-1" (click)="addItem(todoText.value)">
        Add
    </button>
</div>
<table class="table table-striped table-bordered">
    <thead>
        <tr><th></th><th>Description</th><th>Done</th></tr>
    </thead>
    <tbody>
        <tr *ngFor="let item of getTodoItems(); let i = index">
            <td>{{i + 1}}</td>
            <td>{{item.action}}</td>
            <td><input type="checkbox" [(ngModel)]="item.done" /></td>
            <td [ngSwitch]="item.done">
                <span *ngSwitchCase="true">Yes</span>
                <span *ngSwitchDefault>No</span>
            </td>
        </tr>
    </tbody>
</table>
```

У элемента `input` есть атрибут, имя которого начинается с символа `#`; он используется для определения переменной, ссылающейся на элемент в привязках данных шаблона. Эта переменная с именем `todoText` задействована в привязке, примененной к элементу `button`.

```
...
<button class="btn btn-primary m-t-1" (click)="addItem(todoText.value)">
...
```

Этот пример *привязки события* приказывает Angular вызвать метод компонента с именем `addItem`, передавая свойство `value` элемента `input` в аргументе. Листинг 2.15 реализует метод `addItem` в компоненте.

ПРИМЕЧАНИЕ

Пока не беспокойтесь по поводу разных видов привязок. Далее я расскажу о разных типах привязок, которые поддерживаются в Angular, а также о смысле разных типов скобок. Все не так сложно, как может показаться на первый взгляд, особенно когда вы поймете, какое место они занимают в инфраструктуре Angular.

Листинг 2.15. Добавление метода в файл app.component.ts

```
import { Component } from "@angular/core";
import { Model, TodoItem } from "./model";

@Component({
    selector: "todo-app",
    templateUrl: "app.component.html"
})
export class AppComponent {
    model = new Model();

    getName() {
        return this.model.user;
    }

    getTodoItems() {
        return this.model.items.filter(item => !item.done);
    }

    addItem(newItem) {
        if (newItem != "") {
            this.model.items.push(new TodoItem(newItem, false));
        }
    }
}
```

Ключевое слово `import` может использоваться для импортирования нескольких классов из модуля. Одна из команд `import` в листинге была изменена, чтобы класс `TodoItem` мог использоваться в компоненте. В классе компонента метод `addItem` получает текст, отправленный привязкой события в шаблоне, и использует его

для создания нового объекта `TodoItem` и включения его в модель данных. В результате всех этих изменений вы сможете создавать новые задачи: для этого достаточно ввести текст в элементе `input` и нажать кнопку Add (рис. 2.8).

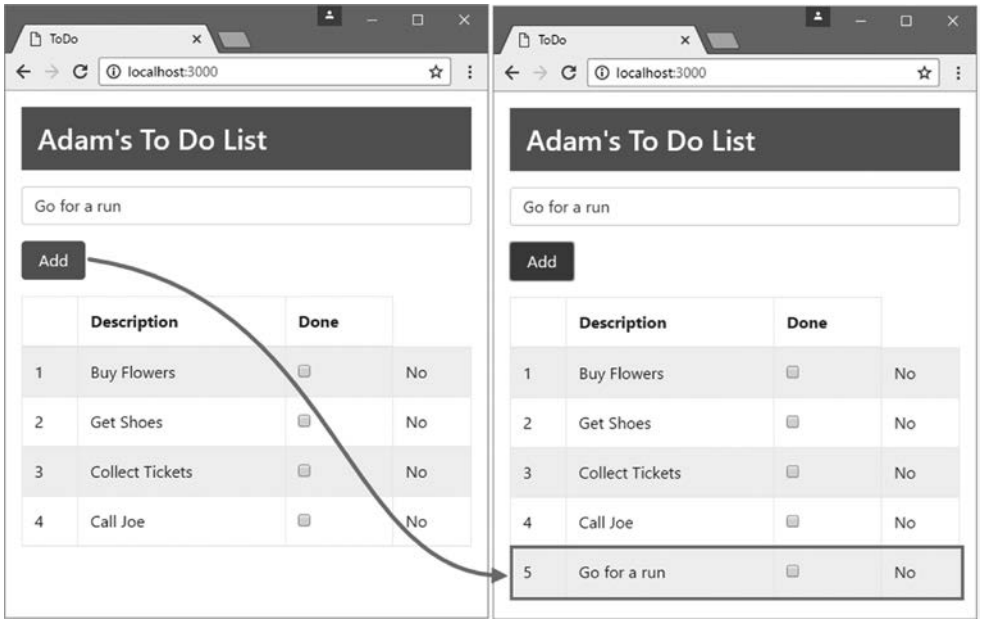


Рис. 2.8. Создание задачи

Итоги

В этой главе я показал, как создать ваше первое простое приложение Angular; от статической разметки HTML мы перешли к динамическому приложению, в котором пользователь может создавать новые задачи и помечать существующие задачи как выполненные.

Не огорчайтесь, если что-то в этой главе осталось непонятным. На этой стадии важнее понять общую структуру приложения Angular, которое строится на основе модели данных, компонентов и шаблонов. Если вы будете постоянно помнить об этих трех ключевых структурных элементах, то у вас появится контекст для понимания всего остального. В следующей главе мы рассмотрим Angular в контексте.

3 Angular в контексте

В этой главе я представлю Angular в контексте разработки веб-приложений и заложу основу для материала последующих глав. Основная цель Angular — открыть веб-клиенту доступ к инструментам и возможностям, которые ранее были доступны только в разработке на стороне сервера, и тем самым упростить разработку, тестирование и сопровождение сложных и полнофункциональных веб-приложений.

Angular позволяет вам *расширять* HTML... Идея может показаться странной, пока вы к ней не привыкнете. Приложения Angular выражают функциональность при помощи нестандартных элементов, а сложное приложение может построить документ HTML, в котором смешивается стандартная и нестандартная разметка.

Стиль разработки, поддерживаемый Angular, основан на использовании паттерна MVC (Model — View — Controller, «модель — представление — контроллер»), хотя его иногда называют «модель — представление — что угодно», потому что существуют бесчисленные вариации этого паттерна, которые могут использоваться с Angular. В книге я сосредоточусь на стандартном паттерне MVC, потому что он был наиболее глубоко проработан и широко применяется на практике. Ниже я опишу характеристики проектов, в которых применение Angular может принести значительную пользу (и тех, для которых существуют более удачные альтернативы), опишу паттерн MVC и некоторые распространенные проблемы при его использовании.

КНИГА И ГРАФИК ВЫПУСКОВ ANGULAR

Компания Google приняла агрессивный план выпуска Angular, начиная с Angular 2.0. Это означает, что дополнительные версии будут появляться постоянно, а основная версия будет выходить каждые полгода. Дополнительные версии не должны нарушать существующую функциональность; в основном они содержат исправления ошибок. Основные версии вводят значительные изменения с возможной потерей обратной совместимости. Согласно этому графику, за версией Angular 2.0, выпущенной в сентябре 2016 года, версии Angular 4.0 и Angular 5.0 последуют в марте и сентябре 2017 года. (Версии 3.0 не будет, а за Angular 2.0 сразу последует Angular 4.0.)

Было бы нечестно и неразумно предлагать читателям покупать новое издание книги через каждые полгода, особенно с учетом того, что основные возможности Angular вряд ли изменятся даже с выходом основной версии. Вместо этого я буду публиковать обновления в репозитории GitHub этой книги (ссылка на него имеется на сайте apress.com).

Такой подход станет экспериментом для меня, и я не знаю, какую форму могут принять такие обновления, еще и потому, что не знаю, что изменится в основных версиях Angular. При этом мы хотим продлить срок жизни книги дополнением содержащихся в ней примеров.

Я не даю никаких обещаний относительно того, как будут выглядеть такие обновления, какую форму они примут и сколько времени я буду работать над ними перед включением в новое издание книги. Пожалуйста, относитесь непредвзято и проверяйте содержимое репозитория книги при выходе новых версий Angular. Если в ходе эксперимента у вас появятся мысли относительно того, как можно улучшить эти обновления, поделитесь ими со мной по адресу adam@adam-freeman.com.

Сильные стороны Angular

Angular не панацея. Важно знать, когда следует применять Angular, а когда лучше поискать альтернативу.

Angular предоставляет функциональность, которая когда-то была доступна исключительно разработчикам на стороне сервера, но теперь требует только наличия браузера. Это означает, что Angular приходится проделывать значительную работу при каждой загрузке документа HTML, к которому применяется Angular: необходимо откомпилировать элементы HTML, обработать привязки данных, выполнить другие структурные элементы и т. д. Все это обеспечивает поддержку тех возможностей, которые были описаны в главе 2 и еще будут описаны в последующих главах книги.

Выполнение такой работы требует времени. Это время зависит от сложности документа HTML, сопутствующего кода JavaScript и — самое важное — от качества браузера и вычислительной мощи устройства. В новейшем браузере на высокопроизводительной настольной машине вы не заметите никаких задержек, но в старом браузере на слабом смартфоне инициализация приложений Angular основательно замедляется.

Таким образом, разработчик должен стараться выполнять инициализацию как можно реже, а если уж она выполняется — предоставить пользователю как можно больше функциональности. Это означает, что вы должны тщательно продумать, какое веб-приложение собираетесь построить. В широком смысле веб-приложения делятся на две категории: *приложения с круговой передачей* (round trip) и *одностраничные приложения* (single-page).

Приложения с круговой передачей и одностраничные приложения

В течение долгого времени веб-приложения строились по модели круговой передачи. Браузер запрашивает у сервера исходный документ HTML. Взаимодействия с пользователем — щелчок на ссылке, отправка данных формы — приводят к тому, что браузер запрашивает и получает совершенно новый документ HTML. В таких приложениях браузер фактически становится средством визуализации контента

HTML, а вся логика и данные приложения размещаются на сервере. Браузер выдает серию запросов HTTP, не имеющих состояния; сервер обрабатывает эти запросы и динамически генерирует документы HTML.

Большая часть современной разработки веб-приложений по-прежнему ориентирована на приложения с круговой передачей. В основном это объясняется тем, что такие приложения почти ничего не требуют от браузера, что гарантирует максимально широкую поддержку клиентов. Однако у приложений с круговой передачей есть серьезные недостатки: они заставляют пользователя ждать, пока будет запрошен и загружен следующий документ HTML; они требуют масштабной инфраструктуры на стороне сервера для обработки всех запросов и управления всем состоянием приложения; они создают повышенную нагрузку на канал, потому что каждый документ должен быть автономным (в результате чего один и тот же контент должен включаться в каждый ответ от сервера).

Одностраничные приложения идут по другому пути. Исходный документ HTML отправляется браузеру, но взаимодействия с пользователем порождают запросы Ajax для получения небольших фрагментов HTML или вставки данных в существующие наборы элементов с последующим выводом. Исходный документ HTML никогда не перезагружается и не замещается; пользователь продолжает взаимодействовать с существующим документом HTML, пока запросы Ajax выполняются асинхронно, даже если взаимодействие сводится к просмотру сообщения «Загрузка данных...».

Многие современные приложения занимают промежуточное место между этими двумя крайностями. Базовая модель круговой передачи, дополненная JavaScript, сокращает количество полных изменений страницы, хотя основное внимание часто уделяется сокращению количества ошибок на формах за счет выполнения проверки на стороне клиента.

Angular обеспечивает наибольшую отдачу от повышения исходных затрат ресурсов тогда, когда приложение приближается к одностраничной модели. Это не означает, что Angular нельзя использовать с приложениями с круговой передачей, — можно, конечно, но существуют и другие технологии, более простые и лучше подходящие для раздельных страниц HTML, основанные либо на выполнении прямых операций с DOM API, либо на использовании библиотек, упрощающих работу с ним (таких, как jQuery). На рис. 3.1 изображена диаграмма типов веб-приложений и область наиболее эффективного применения Angular.

Angular хорошо работает в одностраничных приложениях, и особенно в сложных приложениях с круговой передачей. В более простых проектах лучше использовать прямые операции с DOM API или такие библиотеки, как jQuery, хотя ничто не мешает вам применять Angular во всех ваших проектах.

В современных проектах веб-приложений существует тенденция постепенного перехода на модель одностраничных приложений. Она создает идеальные условия для применения Angular, и не только из-за процесса инициализации, а еще и потому, что преимущества паттерна MVC (о котором я расскажу позднее в этой главе) в полной мере проявляются в более крупных и сложных проектах, которые способствуют переходу на одностраничную модель.

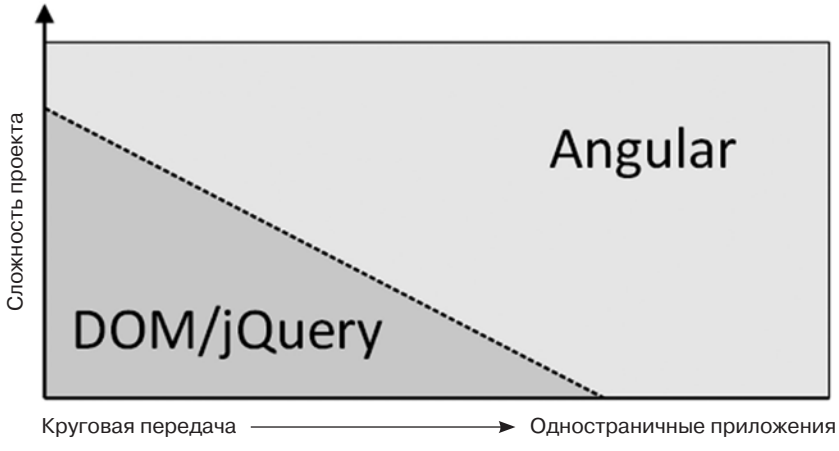


Рис. 3.1. Angular хорошо подходит для модели одностраничных приложений

ПРИМЕЧАНИЕ

Angular и другие похожие фреймворки появились из-за того, что сложные веб-приложения создавали много трудностей в программировании и сопровождении. Проблемы, с которыми сталкивались такие проекты, привели к разработке инструментов промышленного масштаба, таких как Angular. Благодаря этим инструментам появилась возможность создания следующего поколения сложных проектов. Такой вот механизм развития с обратной связью.

ANGULAR И JQUERY

Angular и jQuery используют разные подходы к разработке веб-приложений. Сутью jQuery являются явные манипуляции с браузерной моделью DOM. Angular же привлекает браузер к участию в разработке приложения.

Безусловно, jQuery — эффективный инструмент, которым я люблю пользоваться. Библиотека jQuery мощна и надежна, она позволяет практически сразу же получить результаты. Мне в ней особенно нравится динамичный API и простота расширения базовой библиотеки jQuery. Если вы захотите больше узнать о jQuery, обратитесь к моей книге «Pro jQuery», опубликованной Apress; в ней подробно рассматриваются jQuery, jQuery UI и jQuery Mobile.

Но при всей моей любви к jQuery эта библиотека ничуть не более универсальна, чем Angular. Разработка и сопровождение больших приложений на базе jQuery создают изрядные трудности, как и организация тщательного модульного тестирования.

Angular также использует DOM для представления контента HTML пользователю, но совершенно иначе подходит к построению приложений: основное внимание уделяется данным в приложении и связыванию их с элементами HTML через динамические привязки данных.

Главный недостаток Angular заключается в том, что эта технология требует значительных начальных затрат времени разработки перед тем, как вы увидите первые результаты, — это явление вообще типично для любой разработки на базе MVC. Тем не менее эти исходные затраты оправданны для сложных приложений или приложений, требующих значительных усилий по переработке и сопровождению.

Короче говоря, используйте jQuery (или прямые операции с DOM API) для веб-приложений незначительной сложности, для которых модульное тестирование некритично, и когда вы хотите немедленно получить результаты. Используйте Angular для одностраничных веб-приложений, когда у вас есть время для тщательного планирования и проектирования и когда вы можете легко управлять разметкой HTML, сгенерированной сервером.

Паттерн MVC

Термин *MVC* появился в конце 1970-х годов в ходе работы над одним проектом Smalltalk в Xerox PARC, где эта концепция была предложена для определения структуры первых приложений с графическим интерфейсом. Некоторые аспекты исходного паттерна MVC были связаны с понятиями, специфическими для Smalltalk (такими, как экраны и инструменты), но более широкие идеи применимы к любым приложениям и особенно хорошо подходят для веб-приложений.

Паттерн MVC впервые закрепился на серверной стороне веб-разработки через такие инструментарии, как Ruby on Rails и ASP.NET MVC Framework. За последние годы паттерн MVC также рассматривался как средство управления растущей сложностью и широтой функциональности веб-разработки на стороне клиента. Именно в этой среде появилась технология Angular.

В применении паттерна MVC основную роль играет реализация ключевого принципа *разделения обязанностей*, согласно которому модель данных в приложении отделяется от бизнес-логики и логики представления. В разработке на стороне клиента это означает разделение данных, логики, работающей с этими данными, и элементов HTML, используемых для отображения данных. В результате вы получаете клиентское приложение, более простое в разработке, сопровождении и тестировании.

Три основных структурных блока — *модель*, *контроллер* и *представление*. На рис. 3.2 представлена традиционная структура паттерна MVC применительно к разработке на стороне сервера.

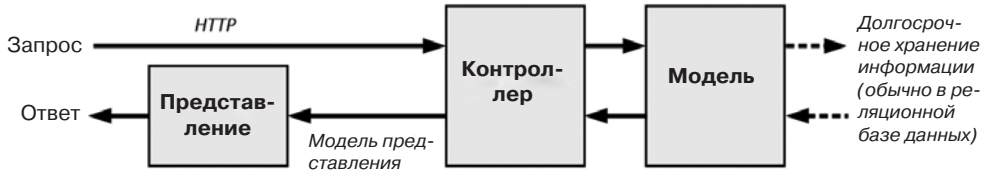


Рис. 3.2. Реализация паттерна MVC на стороне сервера

Я позаимствовал эту диаграмму из своей книги «Pro ASP.NET Core MVC», в которой описана реализация паттерна MVC на стороне сервера от компании Microsoft. Как видно из диаграммы, предполагается, что модель берется из базы данных, а целью приложения является обслуживание запросов HTTP от браузера. Эта схема лежит в основе веб-приложений с круговой передачей, упоминавшихся ранее.

Конечно, Angular существует в браузере, и этот факт оказывает влияние на MVC, как показано на рис. 3.3.

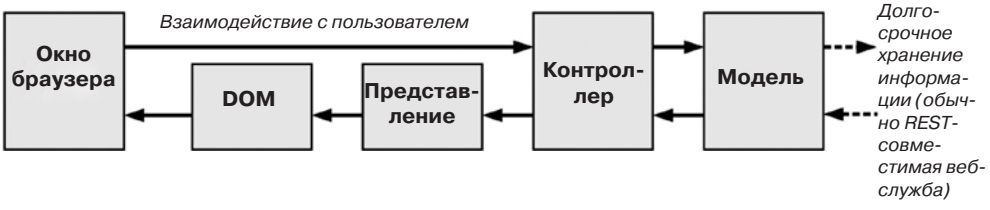


Рис. 3.3. Реализация паттерна MVC на стороне клиента

Реализация паттерна MVC на стороне клиента получает свои данные от компонентов на стороне сервера, обычно через REST-совместимую веб-службу (см. главу 24). Целью контроллера и представления является работа с данными модели и выполнение манипуляций с DOM для создания элементов HTML, с которыми может взаимодействовать пользователь, и управления ими. Взаимодействия передаются обратно контроллеру; таким образом замыкается цикл работы интерактивного приложения.

В Angular используется несколько иная терминология для обозначения структурных блоков. Модель MVC, реализованная средствами Angular, ближе к рис. 3.4.

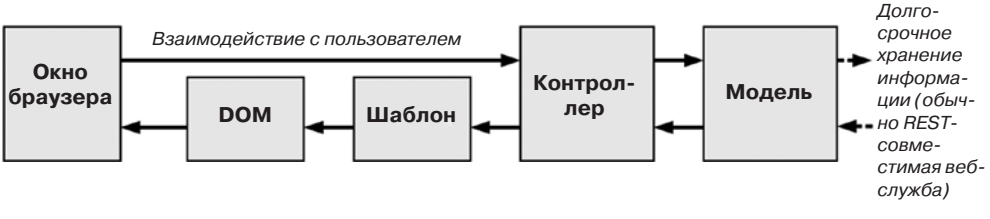


Рис. 3.4. Реализация паттерна MVC в Angular

Диаграмма демонстрирует соответствие между структурными элементами Angular и элементами паттерна MVC. Для поддержки паттерна MVC Angular предоставляет широкий набор дополнительных возможностей, которые будут описаны в книге.

ПРИМЕЧАНИЕ

Использование клиентского фреймворка, такого как Angular, не исключает применения серверного фреймворка MVC, но как вы вскоре увидите, клиент Angular способен взять на себя часть сложности, которая бы в противном случае размещалась на сервере. Обычно это хорошо, потому что нагрузка перемещается с сервера на клиента, что позволяет обслуживать большее количество клиентов при снижении затрат ресурсов сервера.

ПАТТЕРНЫ И ФАНАТИЗМ

Хороший паттерн описывает подход к решению задачи, который хорошо сработал для других людей в других проектах. Паттерны — рецепты, а не правила, и вы должны адаптировать любой паттерн для ваших конкретных проектов подобно тому, как повар адаптирует рецепт для разных печей и доступных ингредиентов.

Степень возможного отклонения от паттерна определяется необходимостью и опытом. Время, которое вы потратили на применение паттерна в похожих проектах, позволит вам понять, что подходит или не подходит для вашей конкретной ситуации. Если вы недавно работаете с паттерном или беретесь за проект нового типа, лучше придерживайтесь паттерна как можно точнее, пока вы не начнете понимать все преимущества и проблемы, которые вас поджидают. Однако не старайтесь переделывать всю разработку «под паттерн», потому что широкомасштабные изменения обычно приводят к потере производительности, а это противоречит тому, чего вы рассчитывали добиться при помощи паттерна.

Паттерны — гибкие инструменты, а не жесткие правила, но не все разработчики понимают это. Некоторые из них относятся к паттернам с излишним фанатизмом; такие люди проводят больше времени за разговорами о паттернах, чем за применением их в проектах, а любое отклонение от их интерпретации паттерна считается серьезным преступлением. Просто не обращайте внимания на таких людей, потому что любые споры только обернутся напрасной тратой усилий, а переубедить их вам все равно не удастся. Вместо этого просто выполните свою работу и покажите, как гибкое применение паттерна позволяет добиться хороших результатов на практике.

Учитывая все сказанное, вы увидите, что в примерах книги я следую общим концепциям паттерна MVC, но при этом адаптирую паттерн для демонстрации различных возможностей и приемов. Именно так я поступаю в своих проектах — задействую те части паттернов, которые приносят пользу в конкретной ситуации, и откладываю в сторону все лишнее.

Модели

Модели — буква «M» в MVC — содержат данные, с которыми работает пользователь. Модели делятся на два типа: *модели представления*, которые представляют данные, передаваемые компонентом шаблону, и *модели предметной области*, которые содержат данные бизнес-области вместе со всеми операциями, преобразованиями и правилами создания, хранения и обработки данных, обозначаемыми общим термином «*логика модели*».

ПРИМЕЧАНИЕ

Многих разработчиков, недавно познакомившихся с паттерном MVC, идея включения логики в модель данных приводит в замешательство: разве целью паттерна MVC не является отделение данных от логики? Нет, это заблуждение: целью архитектуры MVC является разбиение приложения на три функциональные области, каждая из которых может содержать как логику, так и данные. Суть не в том, чтобы полностью устранить логику из модели, а в том, чтобы модель содержала только логику, относящуюся к созданию и управлению данными модели.

В определении паттерна MVC неизбежно привлекает внимание слово «бизнес», и это печально, потому что значительная часть веб-разработки выходит далеко за рамки бизнес-приложений, породивших эту терминологию. Тем не менее бизнес-приложения по-прежнему занимают значительную часть мира разработки, и если вы, скажем, пишете бухгалтерскую систему, ваша бизнес-область будет включать процессы, связанные с ведением бухгалтерского учета, а модель предметной области — данные счетов и логику их создания, хранения и управления. Если вы строите сайт с видеороликами про котиков, бизнес-область присутствует и в этом случае, просто она не вписывается в корпоративную структуру. Модель предметной области в этом случае содержит видеоролики про котиков и логику создания, хранения и управления этими роликами.

Многие модели Angular фактически перемещают логику на сторону сервера и работают с ней через REST-совместимые веб-службы, потому что браузер практически не обеспечивает поддержки долговременного хранения данных и необходимые данные попросту удобнее получить через Ajax. О том, как Angular используется с REST-совместимыми веб-службами, рассказано в главе 24.

ПРИМЕЧАНИЕ

Существуют API хранения данных на стороне клиента, которые были определены в ходе разработки стандарта HTML5. Качество таких стандартов неоднородно, как и качество их реализаций. Однако главная проблема заключается в том, что большинство пользователей продолжает работать с браузерами, не реализующими новые API, и это особенно важно в корпоративных средах, где из-за проблем с переводом бизнес-приложений на стандартные версии HTML до сих пор широко применяются старые версии Internet Explorer. Если вас интересует тема хранения данных на стороне клиента, лучше всего начать со спецификации IndexedDB, которая находится по адресу https://www.w3.org/TR/IndexedDB.

В описании каждого элемента паттерна MVC я укажу, что он должен (или не должен) включать в приложение. Модель в приложении, построенном на базе паттерна MVC, *должна*:

- содержать данные предметной области;
- содержать логику создания, управления и модификации данных предметной области (даже если это означает выполнение удаленной логики через веб-службы);
- предоставлять четкий программный интерфейс (API) для обращения к данным модели и выполнения операций с ними.

Модель *не должна*:

- раскрывать подробности получения данных модели или управления ими (иначе говоря, подробности механизма хранения данных или удаленных веб-служб не должны раскрываться контроллерам и представлениям);
- содержать логику, которая изменяет модель на основании взаимодействий с пользователем (поскольку это задача компонента);
- содержать логику отображения данных для пользователя (поскольку это задача шаблона).

Главное преимущество изоляции такой модели от контроллера и представления — простота тестирования логики (модульное тестирование Angular рассматривается в главе 29), а также простота и удобство расширения и сопровождения всего приложения.

Лучшие доменные модели содержат логику сохранения/загрузки данных, а также логику операций создания, чтения, обновления и удаления (объединяемых сокращением CRUD — Create, Read, Update, Delete) или раздельные модели для обращения с запросами и изменения данных (паттерн CQRS — Command and Query Responsibility Segregation).

Это может означать, что модель содержит логику непосредственного выполнения операций, но чаще модель содержит логику вызова REST-совместимых вебслужб для выполнения операций с базами данных на стороне сервера (которые я продемонстрирую в главе 8, когда построю реалистичное приложение Angular, и опишу подробно в главе 24).

ANGULAR И ANGULARJS

Angular (также Angular 2) — вторая основная версия фреймворка, изначально известного под названием AngularJS. Исходная версия AngularJS была популярной, но неудобной, и разработчикам приходилось иметь дело с запутанными и странно реализованными возможностями, из-за которых веб-разработка была более сложной, чем это необходимо.

Версия Angular, описанная в книге, была полностью переписана и стала более простой в изучении, с ней стало проще работать, а сама она стала более логичной. При этом Angular остается сложным фреймворком, как показывает размер книги, но создавать веб-приложения с Angular намного приятнее, чем с AngularJS.

Различия между AngularJS и Angular настолько принципиальны, что я не стал включать в книгу описания миграции. Если у вас имеется приложение AngularJS, которое вы хотите обновить до Angular, то вы можете воспользоваться адаптером обновления, с которым коды обеих версий фреймворка могут сосуществовать в одном приложении. За подробностями обращайтесь по адресу https://angular.io/docs/ts/latest/guide/upgrade.html. Это может упростить переход, хотя AngularJS и Angular настолько различны, что я бы рекомендовал начать все заново и перейти на Angular с самого начала разработки. Конечно, это не всегда возможно, особенно в сложных приложениях, но процесс миграции с сосуществованием двух версий кода создает большие затруднения и может породить проблемы, которые трудно выявлять и исправлять.

Контроллеры/компоненты

Контроллеры, называемые в Angular *компонентами*, становятся «соединительной тканью» в веб-приложении Angular: они связывают модель данных с представлениями. Компоненты добавляют логику бизнес-области, необходимую для представления некоторого аспекта модели и выполнения операций с ней. Компонент, построенный на базе паттерна MVC, должен:

- содержать логику, необходимую для настройки исходного состояния шаблона;
- содержать логику/поведение, необходимые шаблону для представления данных модели;

- содержать логику/поведение, необходимые для обновления модели на основании взаимодействий с пользователем.

Компонент не должен:

- содержать логику манипуляций с DOM (это задача шаблона);
- содержать логику долгосрочного хранения данных (это задача модели).

Данные представления

Модель предметной области не исчерпывает всех данных в приложениях Angular. Компоненты могут создавать *данные представления* (view data), также называемые *данными модели представления* (view model data) или *моделями представления* (view model), для упрощения шаблонов и их взаимодействий с компонентом.

Представления/шаблоны

Представления, называемые шаблонами в Angular, определяются при помощи элементов HTML, дополненных привязками данных. Именно привязки данных обеспечивают гибкость Angular, и они превращают элементы HTML в основу динамических веб-приложений. Различные типы привязок данных в Angular подробно объясняются в дальнейших главах. Шаблоны должны:

- содержать логику и разметку, необходимые для представления данных пользователю.

Шаблоны не должны:

- содержать сложную логику (которую лучше разместить в компоненте или в одном из других структурных элементов Angular, таких как директивы, службы или каналы);
- содержать логику создания, сохранения или манипуляций с моделью предметной области.

Шаблоны могут содержать логику, но эта логика должна быть простой, а ее объем — умеренным. Размещение в шаблоне чего-либо, кроме простейших вызовов методов или выражений, усложняет тестирование и сопровождение приложения.

REST-совместимые службы

Логика модели предметной области в приложениях Angular часто разбивается между клиентом и сервером. На сервере располагается хранилище информации (обычно база данных) и логика для управления ею. Например, в случае базы данных SQL необходимая логика будет включать открытие подключений к серверу базы данных, выполнение запросов SQL и обработку результатов для отправки клиенту.

Код на стороне клиента не должен обращаться к хранилищу данных напрямую — это привело бы к созданию жесткого сцепления между клиентом и хранилищем данных, которое бы усложнило модульное тестирование и затруднило изменение хранилища данных без внесения изменений в клиентский код.

Использование сервера как посредника в работе с хранилищем данных предотвращает жесткое сцепление. Логика на стороне клиента отвечает за передачу и получение данных от сервера; она изолирована от внутренних подробностей хранения или обращения к данным.

Существует множество способов передачи данных между клиентом и сервером. Один из самых распространенных вариантов — использование запросов Ajax (Asynchronous JavaScript and XML) для вызова кода на стороне сервера, отправка ответов сервером в формате JSON и внесение изменений в данные на формах HTML.

Такой подход может нормально работать, и он лежит в основе REST-совместимых веб-служб, использующих специфику запросов HTTP для выполнения CRUD-операций с данными.

ПРИМЕЧАНИЕ

REST — стиль API, а не формально определенная спецификация, и по поводу того, какая именно веб-служба должна считаться REST-совместимой, существуют разногласия. В частности, пуристы полагают, что веб-службы, возвращающие JSON, не могут считаться REST-совместимыми. Эти разногласия, как и разногласия по поводу любого архитектурного паттерна, субъективны и скучны, и отвлекаться на них не стоит. На мой взгляд, службы JSON являются REST-совместимыми, и в книге я рассматриваю их как таковые.

В REST-совместимой веб-службе запрашиваемая операция выражается комбинацией метода HTTP и URL. Представьте URL следующего вида:

http://myserver.mydomain.com/people/bob

Стандартной спецификации URL-адресов REST-совместимой веб-службы не существует, но идея заключается в том, чтобы сделать URL-адрес по возможности самодокументируемым, чтобы по одному его виду было понятно, что делает URL. В данном случае понятно, что существует коллекция объектов данных с именем `people` и URL ссылается на конкретный объект этой коллекции, идентифицируемый по имени `bob`.

ПРИМЕЧАНИЕ

Создать такие самодокументируемые URL в реальном проекте не всегда возможно, но вы должны по крайней мере серьезно постараться, чтобы не раскрывать внутреннюю структуру хранилища данных через URL (потому что это создает еще одну разновидность сцепления между компонентами). Делайте URL-адреса по возможности простыми и очевидными и сохраните привязку между форматом URL и структурой хранилища данных на сервере.

URL идентифицирует объект данных, с которым выполняется операция, а метод HTTP указывает, какую операцию нужно выполнить. Список методов приведен в табл. 3.1.

Таблица 3.1. Операции, часто выполняемые для различных методов HTTP

Метод	Описание
GET	Возвращает объект данных, заданный URL
POST	Обновляет объект данных, заданный URL
PUT	Создает новый объект данных, обычно с использованием значений данных формы для заполнения полей данных
DELETE	Удаляет объект данных, заданный URL

Для выполнения операций, перечисленных в таблице, не обязательно использовать методы HTTP. Часто метод POST имеет двойное предназначение: он обновляет объект, если он существует, и создает его, если объект не существует; таким образом, метод PUT не используется. Поддержка Ajax и работа с REST-совместимыми службами в Angular рассматриваются в главе 24.

ИДЕМПОТЕНТНЫЕ МЕТОДЫ HTTP

Вы можете реализовать любое соответствие между методами HTTP и операциями с хранилищем данных, хотя я рекомендую по возможности придерживаться схемы, описанной в таблице.

Если вы отклонитесь от нормальной схемы, постарайтесь соблюсти природу методов HTTP, определенную в спецификации HTTP. Метод GET является нульпотентным, то есть операции, выполняемые по этому методу, должны только читать данные, но не изменять их. Браузер (или любой посредник — например, прокси-сервер) ожидает, что повторное выполнение запроса GET не изменит состояние сервера (хотя это не значит, что состояние сервера не изменится между идентичными запросами GET из-за запросов со стороны других клиентов).

Методы PUT и DELETE являются идемпотентными; это означает, что многократные идентичные запросы приводят к такому же эффекту, как и одиночный запрос. Например, использование метода DELETE с URL /people/bob приведет к удалению объекта bob из коллекции people для первого запроса, а для последующих запросов не сделает ничего. (Конечно, если только другой клиент не создаст объект bob заново.)

Метод POST не является ни нульпотентным, ни идемпотентным, поэтому одна из стандартных REST-совместимых оптимизаций направлена на создание и обновление объектов. Если объект bob не существует, метод POST создаст его, а последующие запросы POST для того же URL будут обновлять созданный объект.

Все это важно только в том случае, если вы реализуете собственную REST-совместимую веб-службу. Если вы пишете клиент, использующий готовую REST-совместимую службу, тогда вам достаточно знать, каким операциям данных соответствует каждый метод HTTP. Использование такой службы продемонстрировано в главе 8, а поддержка запросов HTTP в Angular более подробно описана в главе 24.

Распространенные ошибки проектирования

В этом разделе описаны три распространенные ошибки проектирования, встречающиеся в проектах Angular. Это не ошибки программирования, а именно дефекты общей структуры веб-приложения, которые мешают команде проекта в полной мере использовать все преимущества Angular и паттерна MVC.

Неверный выбор места для размещения логики

Самая распространенная проблема — размещение логики не в том компоненте, где она должна находиться, что подрывает разделение обязанностей в MVC. Три наиболее распространенные разновидности этой проблемы:

- размещение бизнес-логики в шаблонах вместо компонентов;
- размещение логики предметной области в компонентах вместо модели;
- размещение логики хранения данных в клиентской модели при использовании REST-совместимой службы.

Все эти проблемы весьма коварны, потому что они не сразу проявляются как проблемы. Приложение работает, но его расширение и сопровождение с течением времени усложняется. В третьем варианте проблема проявляется только при изменении хранилища данных (что происходит редко, пока проект не перерастет исходные оценки).

ПРИМЕЧАНИЕ

Чтобы понять, где должна размещаться та или иная логика, вам потребуется опыт, но модульное тестирование поможет быстрее выявить проблемы, так как тесты, которые вы будете писать для логики, будут плохо укладываться в паттерн MVC. Поддержка модульного тестирование в Angular описана в главе 29.

Умение правильно выбрать место для размещения логики станет вашей второй натурой, когда у вас появится опыт разработки Angular, а пока я приведу три полезных правила:

- Логика шаблона должна подготовить данные только для отображения и никогда — для изменения модели.
- Логика компонента никогда не должна напрямую создавать, обновлять или удалять данные из модели.
- Шаблоны и компоненты никогда не должны напрямую обращаться к хранилищу данных.

Если вы будете помнить об этих правилах в ходе разработки, вам удастся избежать большинства типичных ошибок.

Использование формата хранилища данных

Следующая проблема возникает тогда, когда группа разработки строит приложение, зависящее от специфики хранилища данных на стороне сервера. Недавно

я работал с группой, которая построила своего клиента так, что он учитывал особенности формата данных их сервера базы данных SQL. Но потом возникла проблема (из-за которой меня привлекли к работе): группе потребовалось перейти на более мощную базу данных, которая использовала другое представление ключевых типов данных.

В хорошо спроектированном приложении Angular, которое получает свои данные от REST-совместимой службы, сервер должен скрыть подробности реализации хранилища данных и предоставить клиенту данные в подходящем формате, ориентированном на простоту для клиента. Решите, например, как клиенту потребуется представлять даты, а затем убедитесь в том, что этот формат будет использоваться в хранилище данных, — и если хранилище не поддерживает этот формат во встроенном виде, сервер должен выполнить необходимое преобразование.

Начальные трудности

Angular — сложный фреймворк, который может не раз сбить вас с толку, пока вы к нему не привыкнете. Существует множество разных структурных блоков, которые могут объединяться разными способами для достижения сходных результатов. Этот факт повышает гибкость разработки Angular, а также означает, что вы разработаете собственный стиль решения задач за счет создания функциональных комбинаций, подходящих для проекта и вашего стиля работы.

Однако проблема в том, что освоение Angular требует времени. Возникает искушение взяться за создание собственных проектов еще до того, как вы поймете, как различные части Angular сочетаются друг с другом. Возможно, вы создадите нечто работоспособное, не понимая толком, почему оно работает, — это верный путь к беде, когда вам потребуется что-нибудь изменить. Мой совет — не торопитесь и не жалейте времени, чтобы разобраться во всех возможностях Angular. Никто не запрещает вам создавать проекты с первых дней, но вы должны действительно хорошо понимать, как они работают, и быть готовыми к изменениям, когда вы изучите более эффективные способы достижения своих целей.

Итоги

В этой главе я предоставил контекст для изучения Angular. Вы узнали, в каких проектах стоит применять Angular (а в каких не стоит); как Angular поддерживает паттерн MVC для разработки приложений; я привел краткий обзор стиля REST и его использования для выражения операций с данными через запросы HTTP. Глава завершается описанием трех наиболее распространенных ошибок проектирования в проектах Angular.

В следующей главе приводится краткий курс HTML и CSS-фреймворка Bootstrap, который будет использоваться в примерах книги.

4 Краткий курс HTML и CSS

Разработчики приходят в мир разработки веб-приложений разными путями и не всегда хорошо владеют базовыми технологиями, от которых зависят веб-приложения. В этой главе приводится краткий учебный курс HTML, а также описывается библиотека Bootstrap CSS, которая используется для стилевого оформления примеров книги. В главах 5 и 6 излагаются основы JavaScript и TypeScript вместе с информацией, необходимой для понимания примеров в книге. Опытные разработчики могут пропустить эти вводные главы и перейти сразу к главе 7, в которой я использую Angular для создания более сложного и реалистичного приложения.

Подготовка проекта

В этой главе нам понадобится всего один простой проект. Начните с создания папки с именем HtmlCssPrimer, создайте в ней файл package.json и добавьте контент из листинга 4.1.

Листинг 4.1. Содержимое файла package.json в папке HtmlCssPrimer

```
{
  "dependencies": {
    "bootstrap": "4.0.0-alpha.4"
  },

  "devDependencies": {
    "lite-server": "2.2.2"
  },

  "scripts": {
    "start": "npm run lite",
    "lite": "lite-server"
  }
}
```

Выполните следующую команду из папки HtmlCssPrimer, чтобы загрузить и установить пакеты NPM, указанные в файле package.json:

```
npm install
```

Затем создайте файл index.html в папке HtmlCssPrimer и добавьте разметку из листинга 4.2.

Листинг 4.2. Содержимое файла index.html в папке HtmlCssPrimer

```
<!DOCTYPE html>
<html>
<head>
    <title>ToDo</title>
    <meta charset="utf-8" />
    <link href="node_modules/bootstrap/dist/css/bootstrap.min.css"
          rel="stylesheet" />
</head>
<body class="m-a-1">

    <h3 class="bg-primary p-a-1">Adam's To Do List</h3>

    <div class="m-t-1 m-b-1">
        <input class="form-control" />
        <button class="btn btn-primary m-t-1">Add</button>
    </div>

    <table class="table table-striped table-bordered">
        <thead>
            <tr>
                <th>Description</th>
                <th>Done</th>
            </tr>
        </thead>
        <tbody>
            <tr><td>Buy Flowers</td><td>No</td></tr>
            <tr><td>Get Shoes</td><td>No</td></tr>
            <tr><td>Collect Tickets</td><td>Yes</td></tr>
            <tr><td>Call Joe</td><td>No</td></tr>
        </tbody>
    </table>
</body>
</html>
```

Эта разметка HTML использовалась в главе 2 для моделирования внешнего вида приложения.

Выполните следующую команду из папки HtmlCssPrimer, чтобы запустить сервер HTTP для разработки:

```
npm start
```

Открывается новая вкладка или окно браузера (рис. 4.1).

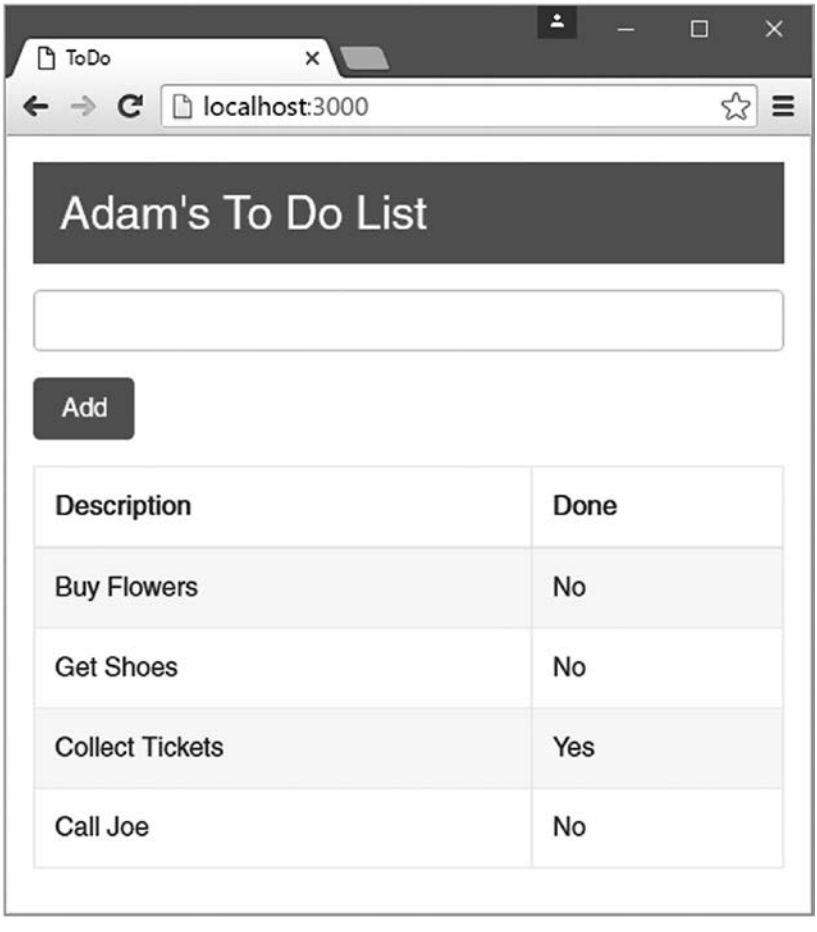


Рис. 4.1. Работающее приложение

Понимание HTML

Центральное место в HTML занимают *элементы*. Они сообщают браузеру, какой вид контента представляет каждая часть документа HTML. Пример элемента из нашего документа:

```
...
<td>Buy Flowers</td>
...
```

Как видно из рис. 4.2, элемент состоит из трех частей: начального тега, конечного тега и контента.

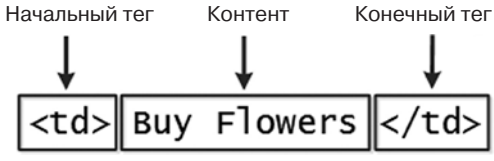


Рис. 4.2. Строение простого элемента HTML

В данном случае *имя* элемента (также называемое именем тега, или просто тегом) имеет вид `td`; оно сообщает браузеру, что контент между тегами следует интерпретировать как ячейку таблицы. Элемент начинается с имени, заключенного в угловые скобки (символы `<` и `>`), и завершается аналогичным тегом, в котором после левой угловой скобки (`<`) добавляется слеш (`/`). Все, что располагается между тегами, — контент элемента. Это может быть как текст (*Buy Flowers* в данном случае), так и другие элементы HTML.

Пустые элементы

В спецификацию HTML включены элементы, которые не могут содержать контент. Такие элементы называются *пустыми*, или *самозакрывающимися*, и записываются без отдельного конечного тега:

```
...
<input />
...
```

Пустой элемент определяется в одном теге, а перед завершающей угловой скобкой (`>`) добавляется символ `/`.

Атрибуты

Вы можете предоставить браузеру дополнительную информацию, добавив атрибуты в элементы. Пример элемента с атрибутами:

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

Этот элемент, определяющий ссылку, импортирует контент в документ. Он содержит два атрибута, выделенные жирным шрифтом. Атрибуты всегда определяются в составе начального тега и состоят из имени и значения.

Имена двух атрибутов в этом примере — `href` и `rel`. У элемента `link` атрибут `href` задает импортируемый контент, а атрибут `rel` сообщает браузеру, к какому типу относится этот контент. Атрибуты элемента `link` приказывают браузеру импортировать файл bootstrap.min.css и интерпретировать его как таблицу стилей — файл, содержащий стили CSS.

Применение атрибутов без значений

Не все атрибуты обязаны иметь значение; сам факт определения атрибута может сообщить браузеру, что с элементом связывается некое поведение. Пример элемента с таким атрибутом:

```
...
<input class="form-control" required />
...
```

Элемент имеет два атрибута. Первому из них — `class` — присваивается значение, как и в предыдущем примере. Второй атрибут состоит из имени `required`; это пример атрибута, которому не нужно присваивать значение.

Литералы в атрибутах

Значительная часть функциональности Angular применяется к элементам HTML посредством атрибутов. В большинстве случаев значения атрибутов вычисляются как выражения JavaScript, как в следующем примере из главы 2:

```
...
<td [ngSwitch]="item.done">
...
```

Атрибут, примененный к элементу `td`, приказывает Angular прочитать значение свойства с именем `done` объекта, присвоенного переменной с именем `item`. В некоторых случаях бывает нужно передать конкретное значение, вместо того чтобы приказывать Angular читать значение из модели данных; такое значение следует заключить в апострофы, чтобы среда Angular понимала, что речь идет о литеральном значении:

```
...
<td [ngSwitch]="'Apples'">
...
```

Значение атрибута содержит строку `Apples`, которая заключается в апострофы и кавычки. При вычислении значения атрибута Angular обнаруживает апострофы и обрабатывает значение как литерал.

Контент элементов

Элементы могут содержать текст, но они также могут содержать другие элементы:

```
...
<thead>
    <tr>
        <th>Description</th>
        <th>Done</th>
    </tr>
</thead>
...
```

Элементы в документе HTML образуют естественную иерархию. Элемент `html` содержит элемент `body`, который в свою очередь содержит элемент `content`; каждый из них может содержать другие элементы, и т. д. В приведенном примере элемент `thead` содержит элементы `tr`, которые в свою очередь содержат элементы `th`. Вложение элементов — одна из ключевых концепций HTML, так как значимость внешнего элемента распространяется на другие элементы, содержащиеся в нем.

Структура документа

Базовая структура любого документа HTML определяется несколькими ключевыми элементами: `DOCTYPE`, `html`, `head` и `body`. Если убрать весь остальной контент, отношение между этими элементами выглядит так:

```
<!DOCTYPE html>
<html>
<head>
    ...Контент head...
</head>
<body>
    ...Контент body...
</body>
</html>
```

Каждый из этих элементов играет определенную роль в документе HTML. Элемент `DOCTYPE` сообщает браузеру, что это документ HTML, а конкретнее — что это документ HTML5. Предыдущие версии HTML требовали дополнительной информации. Например, элемент `DOCTYPE` для документа HTML4 выглядел так:

```
...
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html4/strict.dtd">
...
```

Элемент `html` обозначает область документа, содержащую контент HTML. Этот элемент всегда содержит еще два ключевых структурных элемента: `head` и `body`. Как я объяснял ранее, я не собираюсь описывать отдельные элементы HTML — их слишком много, а полное описание HTML5 заняло более 1000 страниц в моей книге, посвященной HTML. Тем не менее в табл. 4.1 приведены краткие описания элементов, использованных в файле index.html из листинга 4.2. Они помогут вам понять, каким образом элементы передают браузеру информацию о представляемом ими контенте.

МОДЕЛЬ DOM

Когда браузер загружает и обрабатывает документ HTML, он строит модель DOM (Document Object Model). В модели DOM объекты JavaScript представляют каждый элемент документа; этот механизм используется для программного взаимодействия с документом HTML.

В Angular разработчик редко работает с DOM напрямую, но важно понимать, что браузер поддерживает живую модель документа HTML, представленную набором объектов JavaScript. Когда Angular модифицирует эти объекты, браузер обновляет отображаемый контент в соответствии с внесенными изменениями. Это один из ключевых принципов веб-приложений. Если бы мы не могли изменять DOM, то не могли бы и создавать клиентские веб-приложения.

Таблица 4.1. Элементы HTML, использованные в примере

Элемент	Описание
DOCTYPE	Тип контента в документе
body	Область документа, содержащая элементы контента
button	Кнопка; часто используется для отправки данных формы
div	Обобщенный элемент; часто используется для определения структуры документа для целей отображения информации
h3	Заголовок
head	Область документа, содержащая метаданные
html	Область документа, содержащая HTML (обычно весь документ)
input	Поле, предназначенное для ввода одного фрагмента данных пользователем
link	Импортирование содержимого в документе HTML
meta	Информация, описывающая документ (например, кодировка символов)
table	Таблица, контент которой упорядочивается по строкам и столбцам
tbody	Тело таблицы (в отличие от заголовка или завершителя)
td	Ячейка с контентом в строке таблицы
th	Ячейка заголовка в строке таблицы
thead	Заголовок таблицы
title	Название документа; используется браузером для вывода текста в заголовке окна или на вкладке
tr	Строка в таблице

Bootstrap

Элементы HTML сообщают браузеру, какой тип контента они представляют, но не дают никакой информации относительно того, как этот контент должен отображаться. Информация о том, как должны отображаться элементы, содержится в *каскадных таблицах стилей*, или *CSS* (Cascading Style Sheets). CSS состоит из обширного набора свойств, которые позволяют задать каждый аспект внешнего вида элемента, а также набора *селекторов* для применения этих свойств.

Одна из главных проблем с CSS заключается в том, что некоторые браузеры поразному интерпретируют те или иные свойства, что приводит к различиям в спо-

собе отображения контента HTML на разных устройствах. Отслеживать и исправлять такие проблемы нелегко, поэтому появились фреймворки CSS, позволяющие разработчикам веб-приложений просто и последовательно оформлять контент HTML.

Большой популярностью пользуется фреймворк CSS Bootstrap, который изначально был разработан в Twitter, но потом стал популярным проектом с открытым кодом. Bootstrap состоит из набора классов CSS, которые применяются к элементам для их последовательного стилевого оформления, и кода JavaScript, выполняющего дополнительные усовершенствования. Я часто использую Bootstrap в своих проектах — фреймворк прост в использовании и хорошо работает в разных браузерах. Я использую стили Bootstrap в книге, потому что они позволяют оформлять примеры без необходимости определять собственные стили CSS в каждой главе. Bootstrap предоставляет гораздо больше возможностей, чем я использую в книге; за полной информацией обращайтесь по адресу *http://getbootstrap.com*.

ПРЕДВАРИТЕЛЬНАЯ ВЕРСИЯ BOOTSTRAP

Как упоминалось в главе 2, в книге используется предварительная версия фреймворка Bootstrap. Выбирая между версией Bootstrap 3, которая скоро станет устаревшей, и предварительной версией Bootstrap 4, я решил использовать новую версию, несмотря на то что некоторые имена классов, используемых для оформления элементов HTML, могут измениться в окончательной версии. Это означает, что для получения предполагаемых результатов в примерах вы должны использовать ту же версию Bootstrap.

Я не стану слишком углубляться в Bootstrap, потому что книга не об этом, но хочу дать вам достаточно информации, чтобы вы понимали, какие части примера относятся к функциональности Angular, а какие — к стилевому оформлению Bootstrap.

Применение базовых классов Bootstrap

Стили Bootstrap применяются при помощи атрибута `class`, который используется для группировки связанных элементов. Атрибут `class` предназначен не только для применения стилей CSS, но это самое распространенное применение, на котором базируется работа Bootstrap и других похожих фреймворков. Ниже приведен элемент HTML с атрибутом `class` из файла index.html:

```
...
<button class="btn btn-primary m-t-1">Add</button>
...
```

Атрибут `class` относит элемент `button` к трем классам, имена которых разделяются пробелами: `btn`, `btn-primary` и `m-t-1`. Эти классы соответствуют коллекциям стилей, определяемым Bootstrap (табл. 4.2).

Таблица 4.2. Три класса элемента button

Имя	Описание
btn	Класс применяет базовое стилевое оформление кнопок. Он может применяться к button или к элементам a для реализации последовательного оформления
btn-primary	Класс применяет стилевой контекст, предоставляющий визуальные ориентиры относительно назначения кнопки. См. раздел «Контекстные классы»
m-t-1	Класс добавляет отступ между верхом элемента и окружающим его контентом. См. раздел «Поля и отступы»

Контекстные классы

Одно из главных преимущество фреймворков CSS (таких, как Bootstrap) заключается в том, что они упрощают процесс создания единой темы в приложении. Bootstrap определяет набор стилевых контекстов, которые используются для последовательно выдержанного стилевого оформления взаимосвязанных элементов. Эти контексты, описанные в табл. 4.3, используются в именах классов, которые применяют стили Bootstrap к элементам.

Таблица 4.3. Стилевые контексты Bootstrap

Имя	Описание
primary	Контекст для обозначения основного действия или области контента
success	Контекст для обозначения успешного результата
info	Контекст для представления дополнительной информации
warning	Контекст для представления предупреждений
danger	Контекст для представления критических предупреждений
muted	Контекст для снятия смыслового выделения с контента

Bootstrap предоставляет классы, позволяющие применять стилевые контексты а разным типам элементов. В следующем примере контекст `primary` применяется к элементу `h3` из файла index.html, созданного в начале главы:

```
...
<h3 class="bg-primary p-a-1">Adam's To Do List</h3>
...
```

Одним из классов, назначенных этому элементу, является класс `bg-primary`, который назначает цвет фона элемента в соответствии с цветом стилевого контекста. Здесь тот же стилевой контекст применяется к элементу `button`:

```
...
<button class="btn btn-primary m-t-1">Add</button>
...
```

Класс `btn-primary` оформляет элемент `button` или `anchor` с использованием цветов стилевого контекста. Применение одного контекста для оформления разных элементов гарантирует, что они будут выглядеть последовательно и согласованно, как на рис. 4.3: элементы, к которым применяется стилевой контекст, визуально выделяются на фоне других.

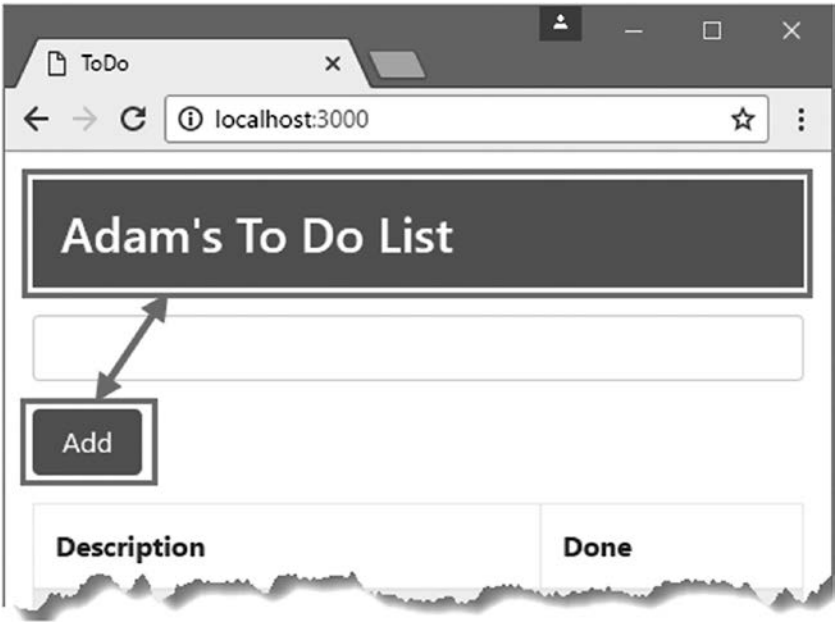


Рис. 4.3. Использование стилевых контекстов для соблюдения единого стиля в оформлении

Поля и отступы

Bootstrap включает набор служебных классов, которые используются для добавления *отступов* — свободного пространства между краем элемента и его контентом и *полей* — пространства между краем элемента и окружающими элементами. Эти классы полезны прежде всего тем, что они применяют в приложении интервалы единого размера.

Имена таких классов строятся по четко определенной схеме. Ниже приведен элемент `body` из файла index.html, созданного в начале главы, с назначением полей:

```
...
<body class="m-a-1">
...
```

Схема формирования имен классов, применяющих поля и отступы к элементам, выглядит так: сначала идет буква `m` (для полей, margin) или `p` (для отступов, padding), затем дефис, буква для выбора сторон(-ы) элемента (`a` — все стороны,

`t` — верхняя сторона, `b` — нижняя сторона, `l` — левая сторона, `r` — правая сторона), снова дефис и цифра, обозначающая величину интервала (`0` — без интервала, `1`, `2` или `3` — последовательно увеличивающиеся интервалы). Чтобы вам было проще понять смысл схемы, в табл. 4.4 перечислены комбинации, используемые в файле index.html.

Таблица 4.4. Примеры классов отступов и полей в Bootstrap

Имя	Описание
p-a-1	Класс применяет отступы ко всем сторонам элемента
m-a-1	Класс применяет поля ко всем сторонам элемента
m-t-1	Класс применяет поля к верхней стороне элемента
m-b-1	Класс применяет поля к нижней стороне элемента

Изменение размеров элементов

Стилевое оформление некоторых элементов можно изменить при помощи класса изменения размера. Имена таких классов строятся из имени базового класса, дефиса и суффикса `lg` или `sm`. В листинге 4.3 я добавил в файл index.html элементы `button` с классами изменения размеров, предоставляемыми Bootstrap для кнопок.

Листинг 4.3. Использование классов изменения размеров кнопок в файле index.html

```
<!DOCTYPE html>
<html>
<head>
    <title>ToDo</title>
    <meta charset="utf-8" />
    <link href="node_modules/bootstrap/dist/css/bootstrap.min.css"
          rel="stylesheet" />
</head>
<body class="m-a-1">

    <h3 class="bg-primary p-a-1">Adam's To Do List</h3>

    <div class="m-t-1 m-b-1">
        <input class="form-control" />
        <button class="btn btn-lg btn-primary m-t-1">Add</button>
        <button class="btn btn-primary m-t-1">Add</button>
        <button class="btn btn-sm btn-primary m-t-1">Add</button>
    </div>

    <table class="table table-striped table-bordered">
        <thead>
            <tr>
                <th>Description</th>
                <th>Done</th>
            </tr>
```

```
        </thead>
        <tbody>
            <tr><td>Buy Flowers</td><td>No</td></tr>
            <tr><td>Get Shoes</td><td>No</td></tr>
            <tr><td>Collect Tickets</td><td>Yes</td></tr>
            <tr><td>Call Joe</td><td>No</td></tr>
        </tbody>
    </table>
</body>
</html>
```

Класс `btn-lg` создает большую кнопку, а класс `btn-sm` — маленькую. Если размер не указан, элементу назначается размер по умолчанию. Обратите внимание на возможность объединения контекстного класса и класса размера.

Сочетание модификаций класса Bootstrap предоставляет полный контроль над стилевым оформлением элементов. Таким образом создается эффект, показанный на рис. 4.4.

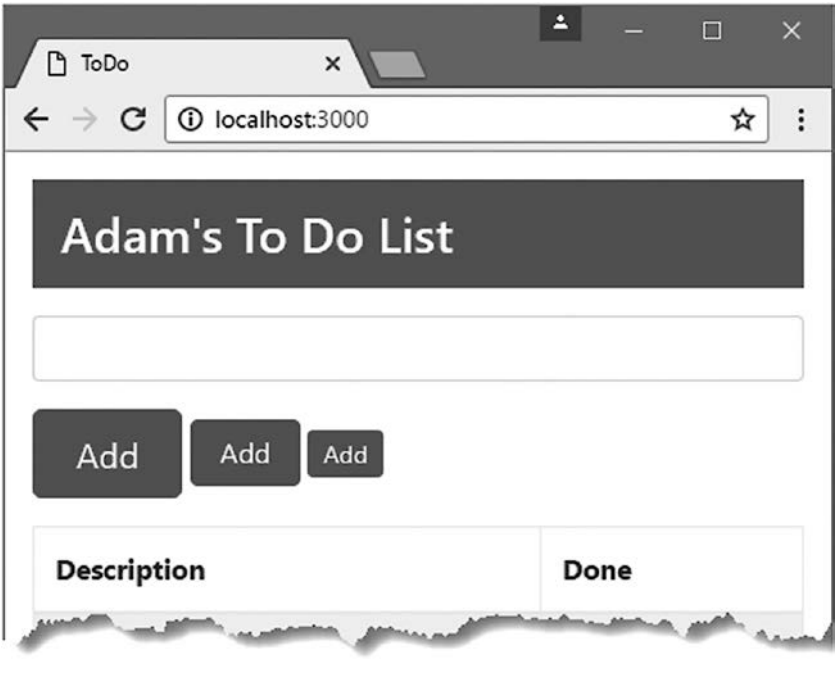


Рис. 4.4. Изменение размера элемента

Использование Bootstrap для оформления таблиц

Bootstrap также поддерживает стилевое оформление таблиц и их содержимого; эта возможность будет часто использоваться в книге. В табл. 4.5 перечислены ключевые классы Bootstrap для работы с таблицами.

Таблица 4.5. Классы Bootstrap для оформления таблиц

Имя	Описание
table	Обобщенное стилевое оформление к элементам table и строкам таблицы
table-striped	Чередование цветов строк в теле таблицы
table-inverse	Инверсия цветов в таблице и ее строках
table-bordered	Применение границ в строках и столбцах таблицы
table-hover	Применение другого стиля при наведении указателя мыши на строку таблицы
table-sm	Сокращение интервалов в таблице для создания более компактного макета

Все эти классы применяются непосредственно к элементу `table`, как показано в листинге 4.4 (классы Bootstrap, применяемые к таблице из файла index.html, выделены жирным шрифтом).

Листинг 4.4. Использование Bootstrap для оформления таблиц

```
...
<table class="table table-striped table-bordered">
    <thead>
        <tr>
            <th>Description</th>
            <th>Done</th>
        </tr>
    </thead>
    <tbody>
        <tr><td>Buy Flowers</td><td>No</td></tr>
        <tr><td>Get Shoes</td><td>No</td></tr>
        <tr><td>Collect Tickets</td><td>Yes</td></tr>
        <tr><td>Call Joe</td><td>No</td></tr>
    </tbody>
</table>
...
```

ПРИМЕЧАНИЕ

Обратите внимание: при определении таблиц в листинге 4.4 используется элемент thead. Браузеры автоматически добавляют в элемент tbody элементы tr, являющиеся прямыми потомками элемента table, если эти элементы не указаны явно. Но если вы будете полагаться на это поведение при работе с Bootstrap, то получите странные результаты, потому что применение большинства классов CSS к элементу table приводит к добавлению стилей к потомкам элемента tbody.

Использование Bootstrap для создания форм

В Bootstrap предусмотрены средства стилевого оформления элементов форм, позволяющие оформлять их в соответствии с другими элементами приложения. В листинге 4.5 я развернул элементы `form` из файла index.html и временно удалил таблицу.

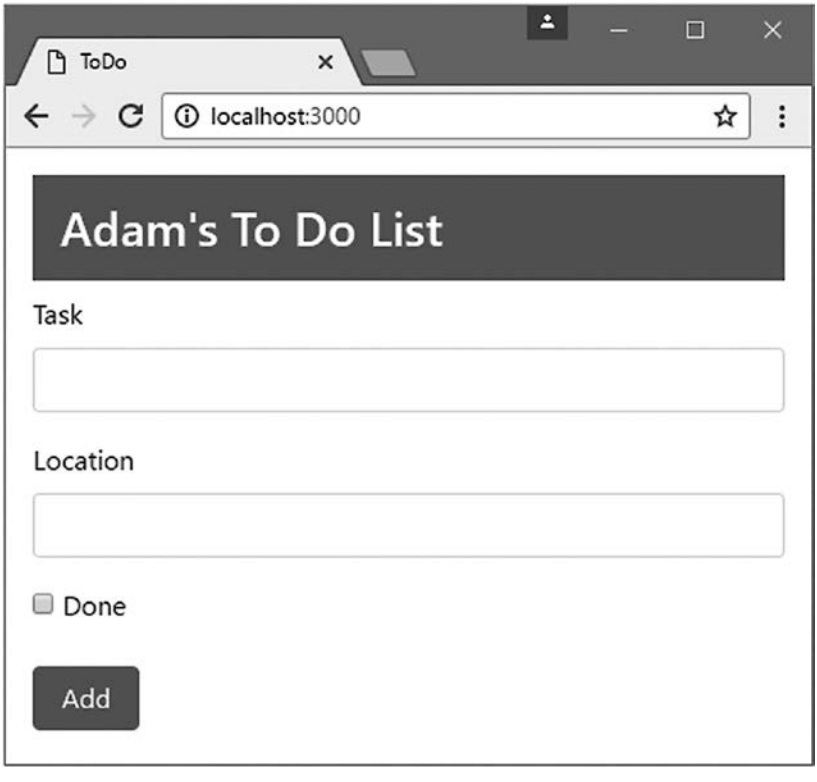


Рис. 4.5. Стилевое оформление элементов формы

Листинг 4.5. Определение дополнительных элементов формы в файле index.html

```
<!DOCTYPE html>
<html>
<head>
    <title>ToDo</title>
    <meta charset="utf-8" />
    <link href="node_modules/bootstrap/dist/css/bootstrap.min.css"
          rel="stylesheet" />
</head>
<body class="m-a-1">
    <h3 class="bg-primary p-a-1">Adam's To Do List</h3>
    <form>
        <div class="form-group">
            <label>Task</label>
            <input class="form-control" />
        </div>
        <div class="form-group">
            <label>Location</label>
            <input class="form-control" />
        </div>
        <div class="form-group">
```

```
            <input type="checkbox" />
            <label>Done</label>
        </div>
        <button class="btn btn-primary">Add</button>
    </form>
</body>
</html>
```

Базовое стилевое оформление форм достигается применением класса `form-group` к элементу `div`, содержащему элементы `label` и `input`; элементу `input` назначается класс `form-control`. Bootstrap оформляет элементы так, что надпись `label` выводится над элементом `input`, а элемент `input` занимает 100% доступного горизонтального пространства (рис. 4.5).

Использование Bootstrap для создания сеток

Bootstrap предоставляет классы стилей, которые могут использоваться для создания различных типов сетчатых макетов, содержащих от 1 до 12 столбцов, и с поддержкой *адаптивных макетов* с изменением макета сетки в зависимости от ширины экрана. В листинге 4.6 контент файла HTML из нашего примера изменен для демонстрации работы с сетками.

Листинг 4.6. Использование средств Bootstrap для работы с сетками в файле index.html

```
<!DOCTYPE html>
<html>
<head>
    <title>ToDo</title>
    <meta charset="utf-8" />
    <link href="node_modules/bootstrap/dist/css/bootstrap.min.css"
          rel="stylesheet" />
    <style>
        #gridContainer {padding: 20px;}
        .row > div { border: 1px solid lightgrey; padding: 10px;
                     background-color: aliceblue; margin: 5px 0; }
    </style>
</head>
<body class="m-a-1">
    <h3 class="panel-header">
        Grid Layout
    </h3>

    <div id="gridContainer">

        <div class="row">
            <div class="col-xs-1">1</div>
            <div class="col-xs-1">1</div>
            <div class="col-xs-2">2</div>
            <div class="col-xs-2">2</div>
            <div class="col-xs-6">6</div>
        </div>
```

```
        <div class="row">
            <div class="col-xs-3">3</div>
            <div class="col-xs-4">4</div>
            <div class="col-xs-5">5</div>
        </div>

        <div class="row">
            <div class="col-xs-6">6</div>
            <div class="col-xs-6">6</div>
        </div>

        <div class="row">
            <div class="col-xs-11">11</div>
            <div class="col-xs-1">1</div>
        </div>

        <div class="row">
            <div class="col-xs-12">12</div>
        </div>
    </div>
</body>
</html>
```

Система сетчатых макетов Bootstrap проста в использовании. Вы указываете столбец, применяя класс `row` к элементу `div`, в результате чего к контенту элемента `div` применяется сетчатый макет. Каждая строка определяет 12 столбцов; чтобы указать, сколько столбцов будет занимать каждый дочерний макет, назначьте класс с именем `col-xs`, за которым следует количество столбцов. Например, класс `col-xs-1` указывает, что элемент занимает один столбец, `col-xs-2` — два столбца и т. д., вплоть до класса `col-xs-12`, который указывает, что элемент занимает всю строку. В листинге я создал серию элементов `div` с классом `row`; каждый элемент содержит другие элементы `div`, к которым были применены классы `col-xs-*`. Результат применения этих классов в браузере показан на рис. 4.6.

ПРИМЕЧАНИЕ

Bootstrap не применяет никакого стилевого оформления к элементам строк, поэтому я использовал элемент style для создания нестандартного стиля CSS, который назначает фоновый цвет, настраивает интервалы между строками и добавляет границу.

Создание адаптивных сеток

Макет адаптивных сеток приспосабливается к размеру окна браузера. Адаптивные сетки используются прежде всего для того, чтобы один и тот же контент мог отображаться на мобильных и настольных устройствах с использованием всей доступной площади экрана. Чтобы создать адаптивную сетку, замените классы `col-*` в отдельных ячейках классами, перечисленными в табл. 4.6.

Если ширина экрана меньше поддерживаемой классом, ячейки строки сетки располагаются по вертикали, а не по горизонтали. Листинг 4.7 демонстрирует создание адаптивной сетки в файле index.html.

Рис. 4.6. Создание сетчатого макета Bootstrap

Таблица 4.6. Классы Bootstrap для создания адаптивных сеток

Класс Bootstrap	Описание
col-sm-*	Ячейки сетки отображаются по горизонтали, если ширина экрана превышает 768 пикселов
col-md-*	Ячейки сетки отображаются по горизонтали, если ширина экрана превышает 940 пикселов
col-lg-*	Ячейки сетки отображаются по горизонтали, если ширина экрана превышает 1170 пикселов

Листинг 4.7. Создание адаптивной сетки в файле index.html

```
<!DOCTYPE html>
<html>
<head>
    <title>ToDo</title>
    <meta charset="utf-8" />
    <link href="node_modules/bootstrap/dist/css/bootstrap.min.css"
          rel="stylesheet" />
    <style>
        #gridContainer {padding: 20px;}
        .row > div { border: 1px solid lightgrey; padding: 10px;
                        background-color: aliceblue; margin: 5px 0; }
    </style>
```

```
</head>
<body class="m-a-1">

    <h3 class="panel-header">
        Grid Layout
    </h3>
    <div id="gridContainer">
            <div class="row">
                <div class="col-sm-3">3</div>
                <div class="col-sm-4">4</div>
                <div class="col-sm-5">5</div>
            </div>

            <div class="row">
                <div class="col-sm-6">6</div>
                <div class="col-sm-6">6</div>
            </div>
            <div class="row">
                <div class="col-sm-11">11</div>
                <div class="col-sm-1">1</div>
            </div>
    </div>
</body>
</html>
```

Я удалил несколько строк сетки в предыдущем примере и заменил классы `col-xs*` классами `col-sm-*`. В результате ячейки строки выстраиваются по горизонтали, если ширина окна браузера превышает 768 пикселов, и по вертикали при меньшей ширине (рис. 4.7).

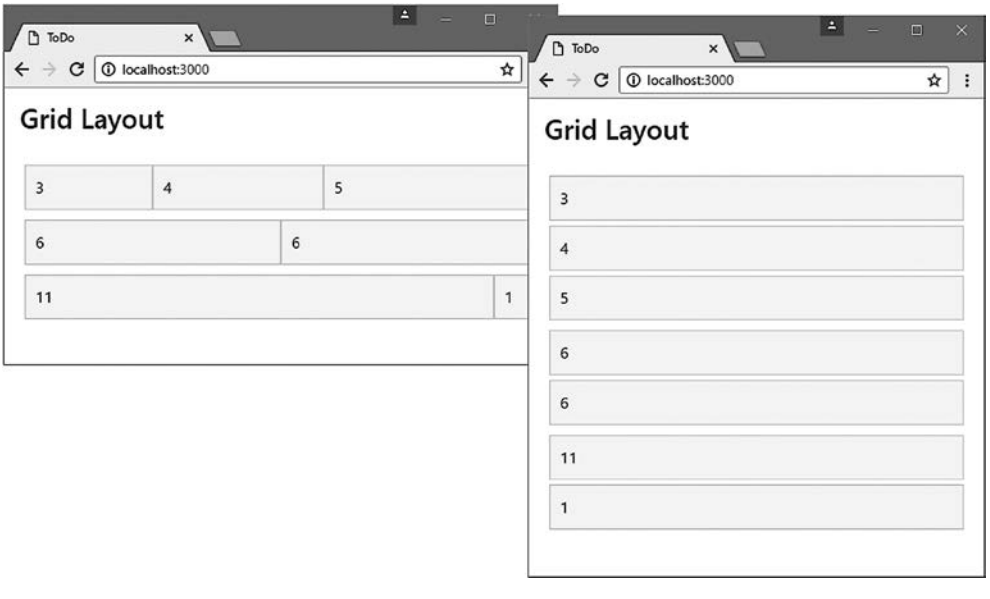


Рис. 4.7. Создание адаптивного сетчатого макета

Создание упрощенного сетчатого макета

В большинстве примеров книги, задействующих сетки Bootstrap, я использую упрощенное решение: контент выводится в одну строку, а задается только количество столбцов, как показано в листинге 4.8.

Листинг 4.8. Использование упрощенного сетчатого макета в файле index.html

```
<!DOCTYPE html>
<html>
<head>
    <title>ToDo</title>
    <meta charset="utf-8" />
    <link href="node_modules/bootstrap/dist/css/bootstrap.min.css"
          rel="stylesheet" />
</head>
<body class="m-a-1">
    <h3 class="bg-primary p-a-1">Adam's To Do List</h3>

    <div class="col-xs-4">
        <form>
            <div class="form-group">
                <label>Task</label>
                <input class="form-control" />
            </div>
            <div class="form-group">
                <label>Location</label>
                <input class="form-control" />
            </div>
            <div class="form-group">
                <input type="checkbox" />
                <label>Done</label>
            </div>
            <button class="btn btn-primary">Add</button>
        </form>
    </div>
    <div class="col-xs-8">
        <table class="table table-striped table-bordered">
            <thead>
                <tr>
                    <th>Description</th>
                    <th>Done</th>
                </tr>
            </thead>
            <tbody>
                <tr><td>Buy Flowers</td><td>No</td></tr>
                <tr><td>Get Shoes</td><td>No</td></tr>
                <tr><td>Collect Tickets</td><td>Yes</td></tr>
                <tr><td>Call Joe</td><td>No</td></tr>
            </tbody>
        </table>
    </div>
</body>
</html>
```

В листинге используются классы `col-xs-4` и `col-xs-8` для отображения двух элементов `div` рядом друг с другом, благодаря чему форма и таблица с задачами размещаются по горизонтали (рис. 4.8).

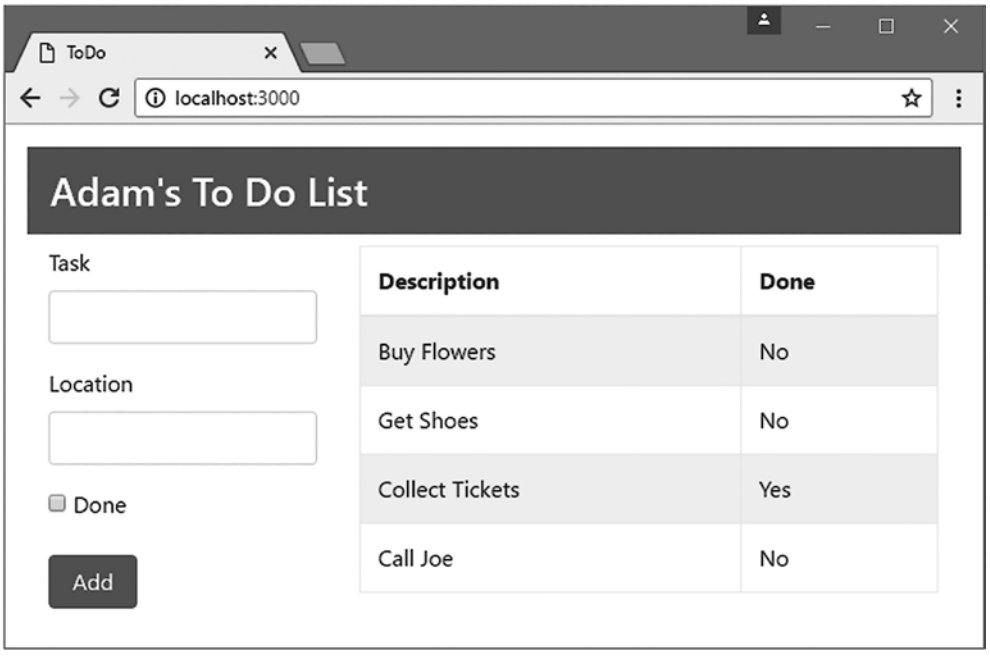


Рис. 4.8. Использование упрощенного сетчатого макета

Итоги

В этой главе приводится краткий обзор HTML и CSS-фреймворка Bootstrap. Чтобы разработка веб-приложений шла действительно эффективно, вы должны хорошо понимать основы HTML и CSS, но учиться лучше всего в ходе практической работы. Описаний и примеров в этой главе будет достаточно, чтобы вы могли взяться за дело, вооружившись необходимой справочной информацией.

В следующей главе наш краткий учебный курс продолжится. Мы рассмотрим базовые возможности JavaScript, используемые в книге.

5 JavaScript и TypeScript: часть 1

В этой главе приведен краткий обзор важнейших базовых возможностей языка JavaScript применительно к разработке Angular. В книге не хватит места для полного описания JavaScript, поэтому я сосредоточился на самом важном, что понадобится для того, чтобы войти в курс дела и разобраться в примерах. В главе 6 описаны более сложные возможности JavaScript, а также дополнительные возможности TypeScript.

Развитие языка JavaScript подчинено стандартному процессу определения новых возможностей. Современные браузеры начали реализовывать стандарт ECMAScript 6 (сокращенно ES6), и на момент написания книги идет процесс определения ECMAScript 7 (ES7). Новые стандарты расширяют функциональность, доступную для разработчиков JavaScript, и приближают JavaScript к таким традиционным языкам, как C# или Java.

Современные браузеры поддерживают автоматическое обновление; это означает, что пользователь Google Chrome, например, с большой вероятностью использует новейшую версию браузера, реализующую по крайней мере часть из современных возможностей JavaScript. К сожалению, старые браузеры, не обновляемые автоматически, все еще распространены достаточно широко, а значит, вы не можете быть уверены в том, что современные возможности будут доступны для вашего приложения.

Проблему можно решить двумя способами. Во-первых, можно использовать только базовые возможности JavaScript, гарантированно присутствующие в браузерах, на которые ориентировано ваше приложение. Во-вторых, можно воспользоваться компилятором, который проводит предварительную обработку файлов JavaScript и преобразует их в код, способный работать в старых браузерах. В Angular используется второй способ, и я опишу его в этой главе. В табл. 5.1 приведена краткая сводка материала главы.

Таблица 5.1. Сводка материала главы

Проблема	Решение	Листинг
Добавление JavaScript в документ HTML	Используйте элемент script или загрузчик модулей JavaScript	1–6
Создание функциональности JavaScript	Используйте команды JavaScript	7

Проблема	Решение	Листинг
Создание групп команд, выполняемых по требованию	Используйте функции	8, 9, 12–14
Определение функций, у которых количество аргументов может быть больше или меньше количества параметров	Используйте значения по умолчанию или остаточные параметры	10–11
Более компактное выражение функций	Используйте лямбда-выражения	15
Хранение значений и объектов для последующего использования	Объявите переменные с ключевым словом let или var	16–17
Хранение базовых значений	Используйте примитивные типы JavaScript	18–21
Управление последовательностью выполнения кода JavaScript	Используйте условные конструкции	22
Проверка совпадения двух значений или объектов	Используйте операторы проверки равенства или тождественности	23–24
Явное преобразование типов	Используйте методы to<тип>	25–27
Совместное хранение взаимосвязанных объектов или значений	Используйте массив	28–33

ИСПОЛЬЗОВАНИЕ «КЛАССИЧЕСКОГО» КОДА JAVASCRIPT С ANGULAR

Приложения Angular можно писать с использованием только тех возможностей JavaScript, которые работают в старых браузерах, без дополнительного этапа обработки файлов компилятором TypeScript. Однако такой код плохо читается и создает много трудностей с сопровождением. В JavaScript появились новые возможности, упрощающие определение и применение средств Angular; одна из них — так называемые декораторы — исключительно важна для эффективной разработки Angular, но это всего лишь предложение для процесса стандартизации JavaScript, которое еще не поддерживается ни одним браузером.

Я рекомендую разобраться во всех нюансах Angular, хотя освоение новых возможностей JavaScript потребует времени и усилий. Результатом станет повышение эффективности разработки и более компактный код, который проще читается и создает меньше трудностей с сопровождением. Этот подход я использовал в книге, и все примеры предполагают, что вы действуете именно так.

В книге не рассматривается тема создания приложений Angular на базе только классических возможностей JavaScript. Если вас интересует именно она, начните с раздела JavaScript на сайте Angular по адресу https://angular.io/docs/js/latest; вы найдете здесь основную информацию, которая поможет вам в изучении темы.

Подготовка примера

Для этой главы я создал новый проект в папке с именем JavaScriptPrimer. Я добавил в папку JavaScriptPrimer файл с именем package.json и включил в него конфигурацию из листинга 5.1.

Листинг 5.1. Содержимое файла package.json в папке JavaScriptPrimer

```
{
  "dependencies": {
    "core-js": "2.4.1",
    "classlist.js": "1.1.20150312",
    "systemjs": "0.19.40",
    "bootstrap": "4.0.0-alpha.4"
  },

  "devDependencies": {
    "lite-server": "2.2.2",
    "typescript": "2.0.3",
    "typings": "1.4.0",
    "concurrently": "3.1.0"
  },

  "scripts": {
    "start": "concurrently \"npm run tscwatch\" \"npm run lite\" ",
    "tsc": "tsc",
    "tscwatch": "tsc -w",
    "lite": "lite-server",
    "typings": "typings"
  }
}
```

Среди пакетов, перечисленных в файле package.json, встречаются библиотеки полизаполнений для поддержки важнейшей функциональности в браузерах со старыми реализациями JavaScript, загрузчик модулей JavaScript и фреймворк Bootstrap. Также включены инструменты, необходимые для использования TypeScript для генерирования файлов JavaScript. Выполните следующую команду из папки JavaScriptPrimer для загрузки и установки пакетов:

```
npm install
```

Создание файлов HTML и JavaScript

Я создал файл с именем primer.ts в папке JavaScriptPrimer и добавил в него код из листинга 5.2. Это всего лишь заполнитель для начала работы над проектом.

ПРИМЕЧАНИЕ

Обратите внимание на расширение файла. Хотя в примерах этой главы используются только возможности JavaScript, они преобразуются компилятором TypeScript в код, работающий в любом браузере. Это означает, что должен использоваться файл с рас-

ширением .ts, чтобы компилятор TypeScript мог создать соответствующий файл .js, работающий в любом браузере.

Листинг 5.2. Содержимое файла primer.ts в папке JavaScriptPrimer

```
console.log("Hello");
```

Я также добавил в папку JavaScriptPrimer файл с именем index.html и разметку HTML, приведенную в листинге 5.3.

Листинг 5.3. Содержимое файла index.html в папке JavaScriptPrimer

```
<!DOCTYPE html>
<html>
<head>
    <title>Primer</title>
        <meta charset="utf-8" />
    <link href="node_modules/bootstrap/dist/css/bootstrap.min.css"
          rel="stylesheet" />
    <script src="primer.js"></script>
</head>
<body class="m-a-1">
    <h3>JavaScript Primer</h3>
</body>
</html>
```

Файл HTML включает элемент `script`, загружающий файл с именем primer.js. Этот файл еще не существует, но компилятор TypeScript сгенерирует его при обработке файла primer.ts.

Настройка компилятора TypeScript

Компилятору TypeScript необходим конфигурационный файл, который указывает, как он должен генерировать файлы JavaScript. Я создал файл с именем tsconfig.json в папке JavaScriptPrimer и добавил данные конфигурации, приведенные в листинге 5.4.

Листинг 5.4. Содержимое файла tsconfig.json в папке JavaScriptPrimer

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "moduleResolution": "node",
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true
  },
  "exclude": [ "node_modules" ]
}
```

Смысл каждого параметра конфигурации описан в главе 11, а пока достаточно просто создать файл.

Выполнение примера

После того как все необходимые файлы будут созданы, выполните следующую команду для запуска компилятора TypeScript и сервера HTTP для разработки:

```
npm start
```

Открывается новая вкладка или окно браузера с содержимым файла index.html (рис. 5.1).

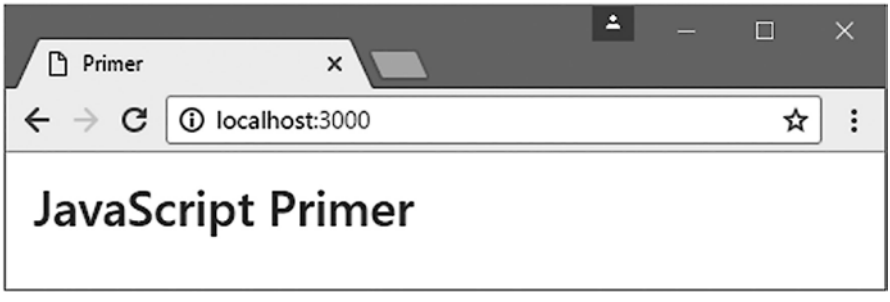


Рис. 5.1. Выполнение примера

Откройте инструментарий разработчика F12 в браузере (он называется так, потому что обычно открывается нажатием клавиши F12) и взгляните на консоль JavaScript (рис. 5.2).

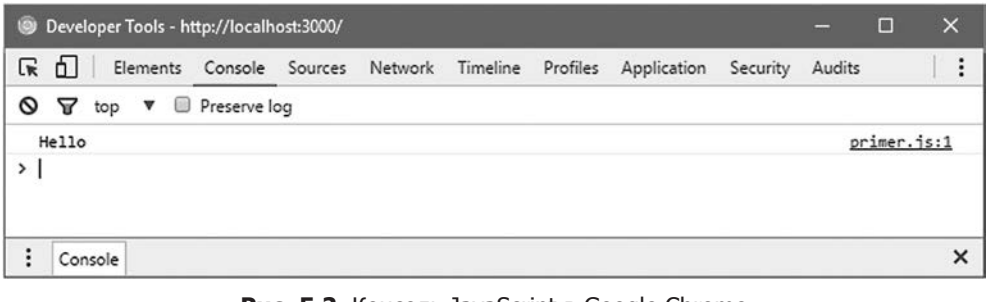


Рис. 5.2. Консоль JavaScript в Google Chrome

На консоли JavaScript выводится результат вызова функции `console.log` из листинга 5.3. Вместо того чтобы приводить снимок экрана с консолью JavaScript в браузере для каждого примера, я просто приведу текст результата:

```
Hello
```

Элемент script

Код JavaScript добавляется в документ HTML элементом `script`. Атрибут `src` указывает, какой файл JavaScript следует загрузить. Еще раз приведу элемент `script` из листинга 5.3:

```
...
<script src="primer.js"></script>
...
```

Элемент задает имя загружаемого файла primer.js. При использовании компилятора TypeScript существует непрямая связь между файлами, содержащими написанный вами код JavaScript, и файлами, загруженными браузером.

Если вы привыкли писать файлы JavaScript напрямую, то переход может создать определенные неудобства, но он позволяет компилятору TypeScript преобразовать многие новые аспекты спецификации JavaScript в код, работающий в старых браузерах.

Использование загрузчика модулей JavaScript

Ручное ведение двух наборов файлов и проверка того, что файл HTML содержит правильный набор элементов `script`, — процесс ненадежный. Например, вы можете добавить элемент `script` для включения файла с расширением .ts (вместо .js) или забудете добавить элемент `script` для нового файла. Также могут возникнуть сложности с порядком элементов `script`. Браузеры выполняют контент файлов JavaScript в порядке, определенном элементами `script` в документе HTML. Легко создать ситуацию, в которой код, загруженный одним элементом `script`, зависит от функциональности, загруженной другим элементом `script` (который еще не был загружен браузером).

Для упрощения управления контентом JavaScript в приложении используется загрузчик модулей, который отвечает за обнаружение и разрешение зависимостей между файлами JavaScript, их загрузку и обеспечение выполнения в правильном порядке. Файл package.json, созданный в начале главы, включает загрузчик модулей SystemJS, а листинг 5.5 показывает, как применить его к документу HTML для управления файлами JavaScript.

Листинг 5.5. Использование загрузчика модулей JavaScript в файле index.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Primer</title>
        <meta charset="utf-8" />
    <link href="node_modules/bootstrap/dist/css/bootstrap.min.css"
          rel="stylesheet" />
    <script src="node_modules/systemjs/dist/system.src.js"></script>
    <script>
        System.config({ packages: {"": {}}});
        System.import("primer").catch(function(err){ console.error(err); });
    </script>
</head>
<body class="m-a-1">
    <h3>JavaScript Primer</h3>
</body>
</html>
```

О том, как создавать модули JavaScript, я расскажу в главе 6, а использование загрузчика модулей SystemJS более подробно рассматривается в главе 11, где я также использую его для загрузки Angular и ряда библиотек, от которых Angular зависит. В этом листинге один элемент `script` загружает загрузчик модулей SystemJS, а другой настраивает и применяет его. Самое важное использование в листинге находится в следующей команде:

```
...
System.import("primer").catch(function(err){ console.error(err); });
...
```

Команда приказывает загрузчику модулей загрузить модуль с именем `primer`; загрузчик преобразует этот приказ в запрос файла primer.js. Одно из преимуществ загрузчика модулей заключается в том, что вам не нужно следить за расширениями файлов в отдельных элементах сценариев.

Основной процесс

Чтобы понять, как разные шаги этого процесса связаны друг с другом, добавьте в файл primer.ts команду из листинга 5.6.

Листинг 5.6. Добавление команды в файл primer.ts

```
console.log("Hello");
console.log("Apples");
```

При сохранении этих изменений в файле primer.ts происходит следующее:

1. Компилятор TypeScript обнаруживает изменения в файле primer.ts и компилирует его для генерирования нового файла primer.js, который может выполняться в любом браузере.
2. Сервер HTTP для разработки обнаруживает изменения в файле primer.js и приказывает браузеру перезагрузить документ HTML.
3. Браузер перезагружает документ HTML и начинает обработку содержащихся в нем элементов. Он загружает файлы JavaScript, указанные в элементах `script` в документе HTML, включая элементы загрузчика модулей JavaScript.
4. Загрузчик модулей JavaScript обрабатывает свою конфигурацию и асинхронно запрашивает файл primer.js у сервера HTTP.
5. Загрузчик модулей JavaScript приказывает браузеру выполнить код в файле pimer.js. Этот код выводит два сообщения на консоль JavaScript браузера.

В результате будут выведены следующие сообщения:

```
Hello
Apples
```

Может показаться, что для такого простого приложения шагов слишком много, но этот подход хорошо масштабируется для сложных проектов и позволяет использовать современные возможности JavaScript.

Чтобы привыкнуть к дополнительным шагам, задействованным в разработке приложений Angular, потребуется время. Процесс разработки может показаться обходным и громоздким, но вскоре он станет вашей второй натурой — особенно когда изменения в коде вызывают автоматическое обновление браузера.

Команды

Основной структурной единицей языка JavaScript является *команда* (statement). Каждая команда представляет некую операцию; обычно команды завершаются символом «точка с запятой» (`;`). Точка с запятой необязательна, но обычно она упрощает чтение кода и позволяет разместить несколько команд в одной строке. В листинге 5.7 я добавил пару команд в файл JavaScript.

Листинг 5.7. Включение команд JavaScript в файл primer.ts

```
console.log("Hello");
console.log("Apples");
console.log("This is a statement");
console.log("This is also a statement");
```

Браузер выполняет каждую команду по порядку. В приведенном примере все команды просто выводят сообщения на консоль. Результат выглядит так:

```
Hello
Apples
This is a statement
This is also a statement
```

Определение и использование функций

Когда браузер получает код JavaScript (напрямую через элемент `script` или косвенно через загрузчик модулей), он выполняет содержащиеся в нем команды в том порядке, в каком они были определены. Именно это происходит в предыдущем примере. Загрузчик модулей загрузил файл primer.js и выполнил содержащиеся в нем команды одну за другой; каждая команда вывела сообщение на консоль.

Команды также можно упаковать в *функцию*, которая будет выполнена лишь после того, как браузер выполнит команду с вызовом этой функции, как показано в листинге 5.8.

Листинг 5.8. Определение функции JavaScript в файле primer.js

```
let myFunc = function () {
    console.log("This is a statement");
};

myFunc();
```

Определить функцию нелсожно: используйте ключевое слово `let` с именем, которое вы хотите присвоить функции, за которым следует знак равенства (`=`), ключевое слово `function` и круглые скобки (символы `(` и `)`). Команды, которые должна содержать функция, заключаются в фигурные скобки (символы `{` и `}`).

В листинге используется имя `myFunc`, а функция состоит из одной команды, которая выводит сообщение на консоль JavaScript. Команда в функции не будет выполняться, пока браузер не встретит строку с вызовом функции `myFunc`:

```
...
myFunc();
...
```

При выполнении этой команды в функции выводится следующее сообщение:

```
This is a statement
```

Это чисто учебный пример определения функции — ведь функция вызывается сразу же после ее определения. Функции приносят гораздо больше пользы тогда, когда они вызываются по некоторому изменению или событию, например при каком-то действии пользователя.

ДРУГОЙ СПОСОБ ОПРЕДЕЛЕНИЯ ФУНКЦИЙ

В JavaScript поддерживаются два способа определения функций. Способ, использованный в листинге 5.8, называется «функциональным выражением». Та же функция может быть определена в следующем виде:

```
...
function myFunc() {
    console.log("This is a statement");
}
...
```

Этот способ называется «объявлением функции». Результат будет тем же: в программе создается функция с именем myFunc, которая выводит сообщение на консоль. Различие в том, как эти функции обрабатываются браузером при загрузке файла JavaScript. Объявления функций обрабатываются перед выполнением кода в файле JavaScript; это означает, что команда с вызовом функции может использоваться до определения самой функции:

```
...
myFunc();
function myFunc() {
    console.log("This is a statement");
}
...
```

Такая конструкция работает, потому что браузер находит объявление функции во время разбора файла JavaScript и создает функцию перед выполнением остальных команд. С функциональными выражениями этого не происходит, поэтому следующий код работать не будет:

```
...
myFunc();

let myFunc = function() {
    console.log("This is a statement");
};
...
```

При попытке выполнения этого кода произойдет ошибка с выдачей сообщения о том, что myFunc не является функцией. Начинающие разработчики JavaScript обычно предпочитают объявления функций, потому что этот синтаксис ближе к синтаксису таких языков, как C# или Java. Выбор зависит исключительно от вас, хотя в проекте рекомендуется использовать единый стиль, чтобы программа была более понятной.

Определение функций с параметрами

JavaScript позволяет определять функции с параметрами. Пример представлен в листинге 5.9.

Листинг 5.9. Определение функций с параметрами в файле primer.ts

```
let myFunc = function(name, weather) {
    console.log("Hello " + name + ".");
    console.log("It is " + weather + " today");
};

myFunc("Adam", "sunny");
```

Я добавил в функцию `myFunc` два параметра с именами `name` и `weather`. JavaScript является языком с динамической типизацией; это означает, что при определении функции не нужно объявлять типы данных ее параметров. Мы вернемся к динамической типизации позже в этой главе, когда речь пойдет о переменных JavaScript. Чтобы вызвать функцию с параметрами, следует передать их значения в аргументах при вызове:

```
...
myFunc("Adam", "sunny");
...
```

Результат вызова функции выглядит так:

```
Hello Adam.
It is sunny today
```

Параметры по умолчанию и остаточные параметры

Количество аргументов, передаваемых при вызове функции, не обязано совпадать с количеством параметров функции. Если при вызове функции будет передано меньше аргументов, чем параметров, то всем отсутствующим параметрам будет присвоено значение `undefined` — специальное значение JavaScript. Если количе-

ство передаваемых аргументов больше количества параметров, лишние аргументы игнорируются.

Из этого следует, что вы не сможете создать две функции с одинаковыми именами и разными наборами параметров и ожидать, что JavaScript сможет различить их по аргументам, переданным при вызове. Такой механизм называется полиморфизмом; хотя он поддерживается в таких языках, как Java и C#, в JavaScript он недоступен. Если вы определяете две функции с одинаковыми именами, то второе определение замещает первое.

Если количество параметров функции в ее определении не соответствует количеству аргументов при ее вызове, возможны два варианта. *Параметры по умолчанию* применяются в ситуации, когда количество аргументов меньше количества параметров; всем параметрам, для которых не указаны аргументы, присваиваются значения по умолчанию, как показано в листинге 5.10.

Листинг 5.10. Использование параметра по умолчанию в файле primer.ts

```
let myFunc = function (name, weather = "raining") {
    console.log("Hello " + name + ".");
    console.log("It is " + weather + " today");
};

myFunc("Adam");
```

Параметру `weather` в функции присваивается значение по умолчанию `raining`, которое используется в том случае, если функция вызывается только с одним аргументом:

```
Hello Adam.
It is raining today
```

Остаточные параметры используются для объединения всех лишних аргументов (листинг 5.11).

Листинг 5.11. Использование остаточного параметра в файле primer.ts

```
let myFunc = function (name, weather, ...extraArgs) {
    console.log("Hello " + name + ".");
    console.log("It is " + weather + " today");
    for (let i = 0; i < extraArgs.length; i++) {
        console.log("Extra Arg: " + extraArgs[i]);
    }
};

myFunc("Adam", "sunny", "one", "two", "three");
```

Остаточный параметр должен быть последним параметром, определяемым функцией, а перед его именем должны стоять три точки (`...`). Остаточный параметр представляет собой массив, элементам которого присваиваются лишние аргументы. В этом примере функция выводит все лишние аргументы на консоль; результат выглядит так:

```
Hello Adam.
It is sunny today
Extra Arg: one
Extra Arg: two
Extra Arg: three
```

Определение функций, возвращающих результаты

Функция может возвращать результат при помощи ключевого слова `return`. В листинге 5.12 приведена функция, возвращающая результат.

Листинг 5.12. Возвращение результата функцией в файле primer.ts

```
let myFunc = function(name) {
    return ("Hello " + name + ".");
};

console.log(myFunc("Adam"));
```

Функция определяет один параметр и использует его для получения результата. Результат вызова передается в аргументе функции `console.log`:

```
...
console.log(myFunc("Adam"));
...
```

Обратите внимание: вам не нужно объявлять, что функция возвращает результат, или указывать тип данных результата. Этот фрагмент выводит следующий результат:

```
Hello Adam.
```

Функции как аргументы других функций

Функции JavaScript могут передаваться как объекты. Это означает, что одна функция может использоваться в качестве аргумента другой функции (листинг 5.13).

Листинг 5.13. Передача функции в аргументе другой функции в файле primer.ts

```
let myFunc = function (nameFunction) {
    return ("Hello " + nameFunction() + ".");
};
console.log(myFunc(function () {
    return "Adam";
}));
```

Функция `myFunc` определяет параметр с именем `nameFunction`, который используется для вставки значения в возвращаемую строку. Я передаю функцию, которая возвращает `Adam`, в аргументе `myFunc`; результат выполнения этого фрагмента выглядит так:

```
Hello Adam.
```

Функции могут объединяться в цепочку. Таким образом из коротких, легко тестируемых фрагментов кода строится более сложная функциональность (листинг 5.14).

Листинг 5.14. Сцепление вызовов функций в файле primer.ts

```
let myFunc = function (nameFunction) {
    return ("Hello " + nameFunction() + ".");
};
let printName = function (nameFunction, printFunction) {
    printFunction(myFunc(nameFunction));
}

printName(function () { return "Adam" }, console.log);
```

Пример выводит такой же результат, как и листинг 5.13.

Лямбда-выражения

Лямбда-выражения (или «функции с толстой стрелкой») предоставляют альтернативный способ определения функций. Они часто применяются для определения функций, которые используются только как аргументы других функций.

В листинге 5.15 функции из предыдущего примера заменяются лямбда-выражениями.

Листинг 5.15. Использование лямбда-выражений в файле primer.ts

```
let myFunc = (nameFunction) => ("Hello " + nameFunction() + ".");

let printName = (nameFunction, printFunction) => printFunction(myFunc(nameFuncti
on));

printName(function () { return "Adam" }, console.log);
```

Эти функции делают то же, что и функции из листинга 5.14. Лямбда-выражения состоят из трех частей: входных параметров, символов `=>` (та самая «толстая стрелка») и результата функции. Ключевое слово `return` и фигурные скобки необходимы только в том случае, если функция должна выполнить более одной команды. Далее в этой главе встречаются несколько примеров лямбда-выражений.

Переменные и типы

Ключевое слово `let` предназначено для объявления переменных с дополнительной возможностью присваивания значения в одной команде. Область действия переменных, объявленных с ключевым словом `let`, ограничивается областью программы, в которой они определяются (листинг 5.16).

Листинг 5.16. Объявление переменных с ключевым словом let в файле primer.ts

```
let messageFunction = function (name, weather) {
    let message = "Hello, Adam";
    if (weather == "sunny") {
        let message = "It is a nice day";
        console.log(message);
    } else {
        let message = "It is " + weather + " today";
        console.log(message);
    }
    console.log(message);
}

messageFunction("Adam", "raining");
```

В этом примере три команды используют ключевое слово `let` для определения переменной с именем `message`. Область действия каждой переменной ограничивается областью кода, в которой эта переменная определяется, поэтому вы получите следующий результат:

```
It is raining today
Hello, Adam
```

Пример может показаться странным, но существует и другое ключевое слово, используемое для объявления переменных: `var`. Ключевое слово `let` было относительно недавно включено в спецификацию JavaScript для исправления некоторых странностей в поведении `var`. Листинг 5.17 берет код из листинга 5.16 и заменяет `let` на `var`.

Листинг 5.17. Объявление переменных с ключевым словом var в файле primer.ts

```
let messageFunction = function (name, weather) {
    var message = "Hello, Adam";
    if (weather == "sunny") {
        var message = "It is a nice day";
        console.log(message);
    } else {
        var message = "It is " + weather + " today";
        console.log(message);
    }
    console.log(message);
}

messageFunction("Adam", "raining");
```

Сохранив изменения в листинге, вы получите следующий результат:

```
It is raining today
It is raining today
```

Проблема в том, что ключевое слово `var` создает переменные, областью видимости которых является функция; это означает, что все ссылки на `message` в функции относятся к одной переменной. Такое поведение может привести к неожиданным

результатам даже для опытных разработчиков JavaScript, поэтому в JavaScript было введено более традиционное ключевое слово `let`. Вы можете использовать в разработке приложений Angular как `let`, так и `var`; в книге используется `let`.

ЗАМЫКАНИЯ ПЕРЕМЕННЫХ

Если функция определяется внутри другой функции (в результате чего появляются две функции, внешняя и внутренняя), но внутренняя функция может обращаться к переменным внешней функции; этот механизм называется замыканием (closure):

```
let myGlobalVar = "apples";
let myFunc = function(name) {
    let myLocalVar = "sunny";
    let innerFunction = function () {
        return ("Hello " + name + ". Today is " + myLocalVar + ".");
    }
    return innerFunction();

};
console.log(myFunc("Adam"));
```

Внутренняя функция в этом примере может обращаться к локальным переменным внешней функции, включая ее параметр. Благодаря этой чрезвычайно полезной возможности вам не придется определять параметры внутренней функции для передачи значений, но будьте внимательны при использовании распространенных имен переменных (таких, как counter или index) — может оказаться, что вы случайно используете имя переменной из внешней функции.

Примитивные типы

JavaScript определяет минимальный набор примитивных типов: `string`, `number` и `boolean`. Список может показаться коротким, но JavaScript наделяет эти три типа выдающейся гибкостью.

Работа с boolean

Тип `boolean` принимает всего два значения: `true` и `false`. В листинге 5.18 приведены примеры использования обоих значений, но этот тип чаще всего используется в условных командах (например, в командах `if`). Консольный вывод в этом листинге отсутствует.

Листинг 5.18. Определение логических значений в файле primer.ts

```
let firstBool = true;
let secondBool = false;
```

Работа со строками

Определяемые значения типа `string` заключаются либо в двойные кавычки, либо в апострофы, как показано в листинге 5.19.

Листинг 5.19. Определение строковых значений в файле primer.ts

```
let firstString = "This is a string";
let secondString = 'And so is this';
```

Ограничители, в которые заключается строка, должны быть парными. Например, нельзя начать строку с апострофа и завершить ее кавычкой. Консольный вывод в этом листинге отсутствует. JavaScript предоставляет базовый набор свойств и методов для объектов `string`, самые полезные из которых перечислены в табл. 5.2.

Таблица 5.2. Свойства и методы string

Имя	Описание
length	Свойство возвращает количество символов в строке
charAt(index)	Метод возвращает строку, содержащую символ с заданным индексом
concat(string)	Метод возвращает новую строку, полученную сцеплением строки, для которой вызывается метод, и строки, переданной в аргументе
indexOf(term, start)	Метод возвращает первый индекс, под которым term встречается в строке, или –1, если подстрока не найдена. Необязательный аргумент start задает начальный индекс при поиске
replace(term, newTerm)	Метод возвращает новую строку, в которой все экземпляры term заменяются на newTerm
slice(start, end)	Метод возвращает подстроку, содержащую символы между индексами start и end
split(term)	Метод разбивает строку в массив значений, разделенных подстрокой term
toUpperCase() toLowerCase()	Метод возвращает новые строки, в которых все символы преобразованы к верхнему или нижнему регистру
trim()	Метод возвращает новую строку, в которой удалены все начальные и конечные символы-пропуски (whitespace)

Строковые шаблоны

Одна из типичных задач программирования — объединение статического содержимого со значениями данных для получения строки, которую можно вывести для пользователя. Традиционно эта задача решается посредством конкатенации строк. Этот метод я до сих пор использовал в примерах этой главы:

```
...
let message = "It is " + weather + " today";
...
```

В JavaScript теперь также поддерживаются *строковые шаблоны* с возможностью подстановки значений данных; этот способ снижает риск ошибок и делает процесс разработки более естественным. В листинге 5.20 представлен пример использования строкового шаблона.

Листинг 5.20. Строковый шаблон в файле primer.js

```
let messageFunction = function (weather) {
    let message = `It is ${weather} today`;
    console.log(message);
}

messageFunction("raining");
```

Строковые шаблоны начинаются и завершаются обратным апострофом (символ `), а значения данных заключаются в фигурные скобки, перед которыми ставится знак `$`. Например, следующая строка подставляет значение переменной `weather` в строковый шаблон:

```
...
let message = `It is ${weather} today`;
...
```

Работа с числами

Тип `number` используется для представления как *целых*, так и *вещественных* чисел (также называемых числами с плавающей точкой). Пример приведен в листинге 5.21.

Листинг 5.21. Определение числовых значений в файле primer.ts

```
let daysInWeek = 7;
let pi = 3.14;
let hexValue = 0xFFFF;
```

Указывать вид используемых чисел не нужно. Вы просто выражаете нужное значение, а JavaScript действует соответствующим образом. В приведенном примере определяется целое значение, определяется значение с плавающей точкой, а также определяется шестнадцатеричное значение с префиксом `0x`.

Операторы JavaScript

JavaScript определяет в целом стандартный набор операторов. Сводка наиболее полезных операторов приведена в табл. 5.3.

Таблица 5.3. Операторы JavaScript

Оператор	Описание
++, --	Префиксный и постфиксный операторы инкремента/декремента
+, -, *, /, %	Сложение, вычитание, умножение, деление, вычисление остатка
<, <=, >, >=	Меньше, меньше либо равно, больше, больше либо равно
==, !=	Проверка равенства и неравенства
===, !==	Проверка тождественности и нетождественности

Оператор	Описание
&&, ||	Логические операции AND и OR (|| используется для слияния null)
=	Присваивание
+	Конкатенация строк
?:	Тернарный условный оператор

Условные команды

Многие операторы JavaScript используются в сочетании с условными командами. В этой книге я обычно использую команды `if/else` и `switch`. В листинге 5.22 продемонстрировано использование обеих команд, которые хорошо знакомы всем, кто работал хотя бы с одним языком программирования.

Листинг 5.22. Использование условных команд if/else и switch в файле primer.ts

```
let name = "Adam";
if (name == "Adam") {
    console.log("Name is Adam");
} else if (name == "Jacqui") {
    console.log("Name is Jacqui");
} else {
    console.log("Name is neither Adam or Jacqui");
}

switch (name) {
    case "Adam":
        console.log("Name is Adam");
        break;
    case "Jacqui":
        console.log("Name is Jacqui");
        break;
    default:
        console.log("Name is neither Adam or Jacqui");
        break;
}
```

Результат выполнения выглядит так:

```
Name is Adam
Name is Adam
```

Оператор равенства и оператор тождественности

Об операторах проверки равенства и тождественности стоит поговорить особо. Первый пытается преобразовать операнды к одному типу для проверки равенства. Это довольно удобно, если вы хорошо понимаете, что такое преобразование выполняется. В листинге 5.23 продемонстрирован оператор проверки равенства в действии.

Листинг 5.23. Оператор проверки равенства в файле primer.ts

```
let firstVal = 5;
let secondVal = "5";

if (firstVal == secondVal) {
    console.log("They are the same");
} else {
    console.log("They are NOT the same");
}
```

Вывод сценария выглядит так:

```
They are the same
```

JavaScript преобразует два операнда к одному типу и сравнивает их. В сущности, оператор проверки равенства проверяет значения на равенство независимо от их типа. Если же вы хотите проверить, что совпадают как значения, так и их типы, используйте оператор тождественности (`===`, три знака равенства вместо двух в операторе проверки равенства), как показано в листинге 5.24.

Листинг 5.24. Использование оператора проверки тождественности в файле primer.ts

```
let firstVal = 5;
let secondVal = "5";

if (firstVal === secondVal) {
    console.log("They are the same");
} else {
    console.log("They are NOT the same");
}
```

В этом примере оператор проверки тождественности считает, что две переменные не тождественны. Оператор не выполняет преобразование типов. Результат выполнения сценария выглядит так:

```
They are NOT the same
```

Явное преобразование типов

Оператор конкатации строк (`+`) имеет более высокий приоритет, чем оператор сложения (тоже `+`); это означает, что JavaScript предпочтет выполнить конкатенацию переменных перед сложением. Это может породить путаницу, потому что JavaScript так же свободно преобразует типы для получения результатов — и результат не всегда соответствует ожиданиям, как показано в листинге 5.25.

Листинг 5.25. Приоритет операторов конкатенации строк в файле primer.ts

```
let myData1 = 5 + 5;
let myData2 = 5 + "5";

console.log("Result 1: " + myData1);
console.log("Result 2: " + myData2);
```

Результат выполнения сценария выглядит так:

```
Result 1: 10
Result 2: 55
```

Именно второй результат может вызвать путаницу. То, что было задумано как операция сложения, интерпретируется как конкатенация строк из-за сочетания приоритета операторов и излишне ретивого преобразования типов. Чтобы избежать этого, можно выполнить явное преобразование типов значений — оно гарантирует, что будет выполнена именно та операция, которая вам нужна (тема преобразования типов более подробно рассматривается ниже).

Преобразование чисел в строки

Если вы работаете с несколькими числовыми переменными и хотите объединить их со строками, вы можете преобразовать числа в строки методом `toString`, как показано в листинге 5.26.

Листинг 5.26. Использование метода number.toString в файле primer.ts

```
let myData1 = (5).toString() + String(5);
console.log("Result: " + myData1);
```

Обратите внимание: числовое значение заключается в круглые скобки, после чего вызывается метод `toString`. Дело в том, что перед вызовом методов, определенных для типа `number`, необходимо преобразовать литерал в число. Я уже показал, как сделать то же самое другим способом — вызовом функции `String` с передачей числового значения в аргументе. Оба способа приводят к одному результату: число преобразуется в строку; следовательно, оператор `+` выполнит конкатенацию строк вместо сложения. Результат выполнения сценария:

```
Result: 55
```

Есть и другие методы, которые позволяют точнее управлять представлением числа в строковом виде. Эти методы кратко описаны в табл. 5.4. Все методы, представленные в таблице, определяются типом `number`.

Таблица 5.4. Полезные методы преобразования чисел в строки

Метод	Описание
toString()	Метод возвращает строку, являющуюся представлением числа в десятеричной системе
toString(2) toString(8) toString(16)	Метод возвращает строку, являющуюся представлением числа в двоичной, восьмеричной или шестнадцатеричной системе
toFixed(n)	Метод возвращает строку, являющуюся представлением вещественного числа с n цифрами в дробной части
toExponential(n)	Метод возвращает строку, являющуюся представлением числа в экспоненциальной записи, с одной цифрой в целой части и n цифрами в дробной части
toPrecision(n)	Метод возвращает строку, являющуюся представлением числа с n значащими цифрами, с применением экспоненциальной записи в случае необходимости

Преобразование строк в числа

Обратное преобразование строк в числа позволяет выполнить сложение вместо конкатенации. Задача решается при помощи функции `Number` (листинг 5.27).

Листинг 5.27. Преобразование строк в числа в файле primer.ts

```
let firstVal = "5";
let secondVal = "5";

let result = Number(firstVal) + Number(secondVal);
console.log("Result: " + result);
```

Результат выполнения сценария выглядит так:

```
Result: 10
```

Функция `Number` выполняет преобразование по строго определенным правилам, но есть еще две функции — `parseInt` и `parseFloat` — более гибкие и игнорирующие завершающие нецифровые символы. Все три метода описаны в табл. 5.5.

Таблица 5.5. Полезные методы преобразования строк в числа

Метод	Описание
Number(str)	Метод разбирает заданную строку и создает целое или вещественное значение
parseInt(str)	Метод разбирает заданную строку и создает целое значение
parseFloat(str)	Метод разбирает заданную строку и создает целое или вещественное значение

Работа с массивами

Массивы JavaScript почти не отличаются от массивов в других языках программирования. В листинге 5.28 приведен пример создания и заполнения массива.

Листинг 5.28. Создание и заполнение массива в файле primer.ts

```
let myArray = new Array();
myArray[0] = 100;
myArray[1] = "Adam";
myArray[2] = true;
```

Новый массив создается вызовом `new Array()`. Вызов создает пустой массив, который присваивается переменной `myArray`. В последующих командах я присваиваю различные значения различным элементам массива. (Консольный вывод в этом листинге отсутствует.)

В этом примере следует обратить внимание на пару моментов. Во-первых, количество элементов в массиве не обязательно указывать при создании. Массивы JavaScript автоматически изменяют размеры под любое количество элементов. Во-вторых, не нужно объявлять тип данных, которые будут храниться в массиве.

Любой массив JavaScript позволяет хранить произвольные комбинации типов данных. В этом примере я присвоил трем элементам массива значения трех разных типов: `number`, `string` и `boolean`.

Литералы массивов

Синтаксис литералов массивов позволяет создать и заполнить массив в одной команде, как показано в листинге 5.29.

Листинг 5.29. Использование литерала массива в файле primer.ts

```
let myArray = [100, "Adam", true];
```

В этом примере я указываю, что переменной `myArray` должен быть присвоен новый массив; для этого элементы, которые должны храниться в массиве, заключаются в квадратные скобки (`[` и `]`). (Консольный вывод в этом листинге отсутствует.)

Чтение и изменение содержимого массива

Чтобы прочитать значение элемента с заданным индексом, заключите индекс в квадратные скобки (листинг 5.30).

Листинг 5.30. Чтение данных из массива в файле primer.ts

```
let myArray = [100, "Adam", true];
console.log("Index 0: " + myArray[0]);
```

Чтобы изменить данные, хранящиеся в любом элементе массива JavaScript, достаточно присвоить новое значение по соответствующему индексу. Как и с обычными переменными, тип данных можно произвольно изменять, и это не создаст никаких проблем. Результат выполнения листинга 5.30 выглядит так:

```
Index 0: 100
```

Изменение содержимого массива продемонстрировано в листинге 5.31.

Листинг 5.31. Изменение содержимого массива в файле primer.ts

```
let myArray = [100, "Adam", true];
myArray[0] = "Tuesday";
console.log("Index 0: " + myArray[0]);
```

В этом примере элементу с индексом 0 в массиве присваивается значение `string`, хотя ранее в этом элементе хранилось значение `number`. Результат:

```
Index 0: Tuesday
```

Перебор элементов массива

Содержимое массива можно перебрать в цикле `for` или при помощи метода `forEach`, который получает функцию, вызываемую для обработки каждого элемента. Оба способа продемонстрированы в листинге 5.32.

Листинг 5.32. Перебор элементов массива в файле primer.ts

```
let myArray = [100, "Adam", true];

for (let i = 0; i < myArray.length; i++) {
    console.log("Index " + i + ": " + myArray[i]);
}

console.log("---");

myArray.forEach((value, index) => console.log("Index " + index + ": " + value));
```

Цикл `for` в JavaScript работает так же, как и циклы во многих других языках. Чтобы узнать текущее количество элементов в массиве, используйте свойство `length`.

Функция, передаваемая методу `forEach`, получает два аргумента: значение текущего обрабатываемого элемента и позиция элемента в массиве. В этом примере я использую лямбда-выражение как аргумент `forEach`; именно в таких ситуациях лямбда-выражения работают особенно эффективно (они еще не раз встретятся вам в книге). Листинг выводит следующий результат:

```
Index 0: 100
Index 1: Adam
Index 2: true

Index 0: 100
Index 1: Adam
Index 2: true
```

Встроенные методы массивов

Объект `Array` в языке JavaScript определяет ряд методов для работы с массивами. Самые полезные из этих методов перечислены в табл. 5.6.

Таблица 5.6. Основные методы массивов

Метод	Описание
concat(otherArray)	Метод возвращает новый массив, повторяющий массив, для которого был вызван, количество раз, заданное в аргументе. При вызове можно передать несколько массивов
join(separator)	Метод объединяет все элементы массива и образует из них строку. Аргумент задает символ, который используется для разделения элементов
pop()	Метод удаляет и возвращает последний элемент массива
shift()	Метод удаляет и возвращает первый элемент массива
push(item)	Метод присоединяет заданный элемент к концу массива
unshift(item)	Метод вставляет новый элемент в начало массива
reverse()	Метод возвращает новый массив с элементами, переставленными в обратном порядке

Метод	Описание
slice(start,end)	Метод возвращает часть массива
sort()	Метод сортирует массив. При вызове можно передать необязательную функцию сравнения для реализации пользовательского порядка сортировки
splice(index, count)	Метод удаляет count элементов из массива, начиная с заданного индекса. Удаленные элементы возвращаются как результат вызова метода
unshift(item)	Метод вставляет новый элемент в начало массива
every(test)	Метод вызывает функцию test для каждого элемента массива; если функция вернула true для каждого элемента, метод возвращает true, в остальных случаях возвращается false
some(test)	Метод возвращает true, если вызов функции test для каждого элемента массива вернул true хотя бы один раз
filter(test)	Метод возвращает новый массив с элементами, для которых функция test возвращает true
find(test)	Метод возвращает первый элемент массива, для которого функция test возвращает true
findIndex(test)	Метод возвращает индекс первого элемента массива, для которого функция test возвращает true
foreach(callback)	Метод вызывает функцию callback для каждого элемента массива (см. предыдущий раздел)
includes(value)	Метод возвращает true, если массив содержит заданный элемент
map(callback)	Метод возвращает новый массив с результатами вызова функции callback для каждого элемента массива
reduce(callback)	Метод возвращает накопленное значение, вычисленное вызовом функции callback для каждого элемента массива

Так как многие методы из табл. 5.6 возвращают новый массив, эти методы можно объединить в цепочку для получения массива данных, как показано в листинге 5.33.

Листинг 5.33. Обработка массива данных в файле primer.ts

```
let products = [
    { name: "Hat", price: 24.5, stock: 10 },
    { name: "Kayak", price: 289.99, stock: 1 },
    { name: "Soccer Ball", price: 10, stock: 0 },
    { name: "Running Shoes", price: 116.50, stock: 20 }
];

let totalValue = products
    .filter(item => item.stock > 0)
    .reduce((prev, item) => prev + (item.price * item.stock), 0);

console.log("Total value: $" + totalValue.toFixed(2));
```

Метод `filter` используется для выбора элементов массива, у которых значение `stock` больше 0, а метод `reduce` вычисляет суммарную стоимость всех позиций. Результат:

```
Total value: $2864.99
```

Итоги

В этой главе приведен краткий вводный курс JavaScript. Основное внимание уделяется базовой функциональности, которая поможет вам взяться за работу с языком. Некоторые возможности, описанные в этой главе, были относительно недавно включены в спецификацию JavaScript, и для преобразования их в код, работающий в старых браузерах, потребуется компилятор TypeScript. Эта тема продолжается в следующей главе, в которой будут рассмотрены некоторые расширенные возможности JavaScript, применяемые в разработке приложений Angular.

6 JavaScript и TypeScript: часть 2

В этой главе будут представлены некоторые расширенные возможности JavaScript, применяемые в разработке Angular. Я объясню, как JavaScript работает с объектами (включая поддержку классов) и как функциональность JavaScript упаковывается в модули JavaScript. Также будут представлены некоторые возможности, предоставляемые компилятором TypeScript и не входящие в спецификацию JavaScript; они встречаются в некоторых примерах книги. В табл. 6.1 приведена краткая сводка материала главы.

Таблица 6.1. Сводка материала главы

Проблема	Решение	Листинг
Создание объекта с заданием его свойств	Используйте ключевое слово new или объектный литерал	1–3
Создание объекта по шаблону	Определите класс	4, 5
Наследование поведения от другого класса	Используйте ключевое слово extends	6
Группировка функциональности JavaScript	Создайте модуль JavaScript	7
Объявление зависимости в модуле	Используйте ключевое слово import	8–12
Объявление типов свойств, параметров и переменных	Используйте аннотации TypeScript	13–18
Определение множественных типов	Используйте объединенные типы	19–20
Группировка разнотипных значений	Используйте кортежи	21
Группировка значений по ключу	Используйте индексируемые типы	22
Управление доступом к методам и свойствам класса	Используйте модификаторы управления доступом	23

Подготовка примера

В этой главы мы продолжим использовать проект JavaScriptPrimer из главы 5. Никакие изменения не потребуются, а выполнение следующей команды из папки JavaScriptPrimer запустит компилятор TypeScript и сервер HTTP для разработки:

```
npm start
```

Откроется новое окно браузера с контентом, показанным на рис. 6.1.

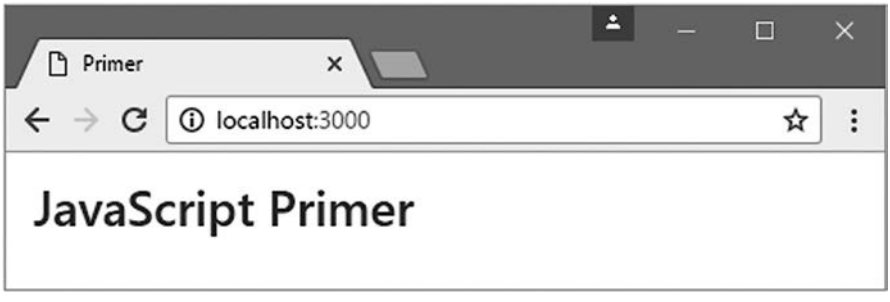


Рис. 6.1. Выполнение примера

Примеры этой главы используют консоль JavaScript в браузере для вывода сообщений. Взглянув на консоль, вы увидите следующий результат:

```
Total value: $2864.99
```

Работа с объектами

В JavaScript поддерживаются разные способы создания объектов. В листинге 6.1 представлен простой пример.

ПРИМЕЧАНИЕ

В некоторых примерах этой главы компилятор TypeScript выводит сообщения об ошибке. Примеры все равно работают, и на сообщения можно не обращать внимания; они появляются из-за того, что TypeScript предоставляет дополнительные возможности, которые будут описаны позже в этой главе.

Листинг 6.1. Создание объекта в файле primer.ts

```
let myData = new Object();
myData.name = "Adam";
myData.weather = "sunny";

console.log("Hello " + myData.name + ".");
console.log("Today is " + myData.weather + ".");
```

Объект создается вызовом `new Object()`, после чего результат (созданный объект) присваивается переменной с именем `myData`. После того как объект будет создан, его свойства определяются простым присваиванием значений:

```
...
myData.name = "Adam";
...
```

До выполнения этой команды у объекта не было свойства `name`. После выполнения команды такое свойство существует, и ему присвоено значение `Adam`. Чтобы

прочитать значение свойства, укажите имя переменной и имя свойства через точку:

```
...
console.log("Hello " + myData.name + ".");
...
```

Результат выглядит так:

```
Hello Adam.
Today is sunny.
```

Объектные литералы

Определить объект и его свойства можно за один шаг. Для этого используется формат объектных литералов, продемонстрированный в листинге 6.2.

Листинг 6.2. Использование формата объектного литерала в файле primer.ts

```
let myData = {
    name: "Adam",
    weather: "sunny"
};

console.log("Hello " + myData.name + ". ");
console.log("Today is " + myData.weather + ".");
```

Имя каждого определяемого свойства отделяется от его значения двоеточием (`:`), а свойства разделяются запятыми (`,`). Этот синтаксис приводит к тому же результату, что и предыдущий пример; на консоль будут выведены следующие сообщения:

```
Hello Adam.
Today is sunny.
```

Функции как методы

Одна из самых полезных возможностей JavaScript — добавление функций к объектам. Функция, определяемая для объекта, называется *методом*. В листинге 6.3 показано, как добавлять методы к объекту.

Листинг 6.3. Добавление методов к объекту в файле primer.ts

```
let myData = {
    name: "Adam",
    weather: "sunny",
    printMessages: function () {
        console.log("Hello " + this.name + ". ");
        console.log("Today is " + this.weather + ".");
    }
};
myData.printMessages();
```

В этом примере функция используется для создания метода с именем `printMessages`. Обратите внимание: для обращения к свойствам, определяемым объектом, используется ключевое слово `this`. При использовании функции как метода функции объект, для которого был вызван метод, неявно передается в специальной переменной `this`. Вывод листинга выглядит так:

```
Hello Adam.
Today is sunny.
```

Определение классов

Классы представляют собой шаблоны, используемые для создания объектов с идентичной функциональностью. Поддержка классов, недавно включенная в спецификацию JavaScript, должна была привести JavaScript в соответствие с другими популярными языками программирования; она широко применяется в разработке Angular. Листинг 6.4 показывает, как функциональность, определяемая объектом из предыдущего раздела, может выражаться с использованием класса.

Листинг 6.4. Определение класса в файле primer.ts

```
class MyClass {

    constructor(name, weather) {
        this.name = name;
        this.weather = weather;
    }

    printMessages() {
        console.log("Hello " + this.name + ". ");
        console.log("Today is " + this.weather + ".");
    }
}

let myData = new MyClass("Adam", "sunny");
myData.printMessages();
```

Классы JavaScript знакомы всем программистам, работавшим на любом из популярных языков (таких, как Java или C#). Ключевое слово `class` используется для объявления класса; за ним следует имя класса, в данном случае `MyClass`. Конструктор — функция, вызываемая при создании нового объекта на основе класса, — предоставляет возможность получить данные и провести инициализацию, необходимую для класса. В примере конструктор определяет параметры `name` и `weather`, которые используются для создания одноименных переменных. Переменные, определяемые таким образом, называются *свойствами* (properties).

Классы также могут содержать методы, которые определяются как функции, но без обязательного использования ключевого слова `function`. В этом примере представлен один метод с именем `printMessages`, который использует значения свойств `name` и `weather` для вывода сообщений на консоль JavaScript в браузере.

ПРИМЕЧАНИЕ

Классы также могут содержать статические методы, помеченные ключевым словом static. Статические методы принадлежат классу, а не объектам, созданным на основе этих классов. Пример статического метода встречается в листинге 6.14.

Ключевое слово `new` используется для создания объекта на основе класса:

```
...
let myData = new MyClass("Adam", "sunny");
...
```

Команда создает новый объект, при этом класс `MyClass` используется в качестве шаблона. `MyClass` используется в этой ситуации как функция, а переданные аргументы будут получены функцией-конструктором, определяемой классом. Результатом этого выражения является новый объект, который присваивается переменной с именем `myData`. После того как объект будет создан, вы сможете обращаться к его свойствам и методам через переменную, которой объект был присвоен:

```
...
myData.printMessages();
...
```

Пример выводит на консоль JavaScript в браузере следующий результат:

```
Hello Adam.
Today is sunny.
```

КЛАССЫ JAVASCRIPT И ПРОТОТИПЫ

Поддержка классов не изменяет основополагающий механизм работы с типами в JavaScript. Вместо этого классы просто предоставляют способ использования типов, более знакомый большинству программистов. Во внутренней реализации JavaScript продолжает использовать традиционную систему типов, основанную на прототипах. Например, код из листинга 6.4 можно записать так:

```
var MyClass = function MyClass(name, weather) {
    this.name = name;
    this.weather = weather;
}

MyClass.prototype.printMessages = function () {
    console.log("Hello " + this.name + ". ");
    console.log("Today is " + this.weather + ".");
};

var myData = new MyClass("Adam", "sunny");
myData.printMessages();
```

Классы упрощают разработку приложений Angular, поэтому в книге я использовал именно этот способ. Многие возможности, появившиеся в ES6, относятся к категории «синтаксических удобств»; иначе говоря, они упрощают понимание и использование некоторых аспектов JavaScript. У JavaScript есть свои странности, и многие из «синтаксических удобств» помогают разработчику избежать типичных ошибок.

Определение свойств с get- и set-методами

Классы JavaScript могут определять свойства в конструкторе; в результате создается переменная, которую можно читать и изменять в любой точке приложения. Свойства с get- и set-методами выглядят за пределами класса как обычные свойства, но они позволяют ввести дополнительную логику, которая может использоваться для проверки или преобразования новых значений или генерирования их на программном уровне, как показано в листинге 6.5.

Листинг 6.5. Get- и set-методы в файле primer.ts

```
class MyClass {
    constructor(name, weather) {
        this.name = name;
        this._weather = weather;
    }

    set weather(value) {
        this._weather = value;
    }

    get weather() {
        return `Today is ${this._weather}`;
    }

    printMessages() {
        console.log("Hello " + this.name + ". ");
        console.log(this.weather);
    }
}

let myData = new MyClass("Adam", "sunny");
myData.printMessages();
```

Get- и set-методы реализуются в виде функций, перед которыми ставится ключевое слово `get` или `set`. В классах JavaScript отсутствует понятие уровней доступа, а по общепринятой схеме имена внутренних свойств начинаются с символа подчеркивания (`_`). В листинге свойство `weather` реализуется set-методом, который обновляет внутреннее свойство с именем `_weather`, и get-методом, который встраивает значение `_weather` в строковый шаблон.

Пример выводит следующий результат на консоль JavaScript в браузере:

```
Hello Adam.
Today is sunny
```

Наследование

Классы могут наследовать поведение от других классов при помощи ключевого слова `extends`, как показано в листинге 6.6.

Листинг 6.6. Наследование в файле primer.ts

```
class MyClass {
    constructor(name, weather) {
```

```
        this.name = name;
        this._weather = weather;
    }

    set weather(value) {
        this._weather = value;
    }

    get weather() {
        return `Today is ${this._weather}`;
    }

    printMessages() {
        console.log("Hello " + this.name + ". ");
        console.log(this.weather);
    }
}

class MySubClass extends MyClass {

    constructor(name, weather, city) {
        super(name, weather);
        this.city = city;
    }

    printMessages() {
        super.printMessages();
        console.log(`You are in ${this.city}`);
    }
}

let myData = new MySubClass("Adam", "sunny", "London");
myData.printMessages();
```

Ключевое слово `extends` используется для объявления класса, от которого наследует определяемый класс; этот класс называется *суперклассом*, или *базовым классом*. В данном примере `MySubClass` наследует от `MyClass`. Ключевое слово `super` используется для вызова конструктора и методов суперкласса. `MySubClass` использует функциональность `MyClass` и дополняет ее поддержкой `city`. На консоль JavaScript в браузере выводится следующий результат:

```
Hello Adam.
Today is sunny
You are in London
```

Работа с модулями JavaScript

Модули JavaScript используются для управления зависимостями в веб-приложениях; это означает, что вам не придется управлять набором элементов `script` в документе HTML. Вместо этого загрузчик модулей сам определяет, какие файлы потребуются для выполнения приложения, загружает эти файлы и выполняет их в правильном порядке. В сложном приложении решать эту задачу вручную достаточно тяжело, и она особенно хорошо подходит для автоматизации.

Создание модулей

Создать модули несложно; более того, это происходит автоматически, когда TypeScript компилирует файлы, потому что каждый файл интерпретируется как модуль. Ключевым словом `export` обозначаются переменные и классы, которые могут использоваться за пределами файла; это означает, что другие переменные и классы могут использоваться только внутри файла. Чтобы показать, как работают модули, я создал папку с именем modules в папке JavaScriptPrimer, добавил в нее файл с именем NameAndWeather.ts и использовал его для определения классов из листинга 6.7.

ПРИМЕЧАНИЕ

В Angular принята схема с определением одного класса на файл; это означает, что каждый класс в проекте Angular содержится в отдельном модуле, и ключевое слово export часто встречается в примерах книги.

Листинг 6.7. Содержимое файла NameAndWeather.ts в папке JavaScriptPrimer/modules

```
export class Name {
    constructor(first, second) {
        this.first = first;
        this.second = second;
    }

    get nameMessage() {
        return `Hello ${this.first} ${this.second}`;
    }
}

export class WeatherLocation {
    constructor(weather, city) {
        this.weather = weather;
        this.city = city;
    }

    get weatherMessage() {
        return `It is ${this.weather} in ${this.city}`;
    }
}
```

Ключевое слово `export` применяется к каждому из классов нового файла; это означает, что они могут использоваться в любой точке приложения.

РАЗНЫЕ ФОРМАТЫ МОДУЛЕЙ

Модули JavaScript делятся на несколько типов, каждый из которых обслуживается отдельным набором загрузчиков модулей. Компилятор TypeScript может быть настроен для генерирования самых популярных форматов, которые задаются свойством module в файле tsconfig.json. Для разработки Angular используется формат commonjs, который встречается в примере файла tsconfig.json из главы 5.

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "moduleResolution": "node",
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true
  },
  "exclude": [ "node_modules" ]
}
```

Для этого формата подходит загрузчик модулей SystemJS, который используется в примерах книги; его описание приводится в главе 11.

Импортирование из модулей JavaScript

Ключевое слово `import` используется для объявления зависимостей для содержимого модуля. Есть несколько вариантов использования ключевого слова `import`, описанных в следующих разделах.

Импортирование конкретных типов

Традиционно ключевое слово `import` используется для объявления зависимостей от конкретных типов (листинг 6.8). Такой способ сводит к минимуму количество зависимостей в приложении, что позволяет оптимизировать приложение при развертывании (см. главу 10).

Листинг 6.8. Импортирование конкретных типов в файле primer.ts

```
import { Name, WeatherLocation } from "./modules/NameAndWeather";

let name = new Name("Adam", "Freeman");
let loc = new WeatherLocation("raining", "London");

console.log(name.nameMessage);
console.log(loc.weatherMessage);
```

Этот способ использования ключевого слова `import` в большинстве примеров книги. За ключевым словом следуют фигурные скобки с разделенным запятыми списком классов, от которых зависят текущие файлы; за ними следует ключевое слово `from` и имя модуля. В данном случае импортируются классы `Name` и `WeatherLocation` из модуля `NameAndWeather` в папке modules. Обратите внимание: расширение файлов при указании модуля не указывается.

ПРИМЕЧАНИЕ

Обратите внимание: я не вношу изменения в проект, чтобы использовать модуль NameAndWeather. Команда import в файле primer.ts объявляет зависимость от модуля, и эта зависимость автоматически разрешается загрузчиком модулей.

Пример из листинга 6.8 выводит на консоль JavaScript в браузере следующие сообщения:

```
Hello Adam Freeman
It is raining in London
```

РАЗРЕШЕНИЕ ЗАВИСИМОСТЕЙ МОДУЛЕЙ

В этой книге встречаются два разных способа задания модулей в командах import. В первом варианте имя модуля снабжается префиксом ./, как в примере из листинга 6.8:

```
...
import { Name, WeatherLocation } from "./modules/NameAndWeather";
...
```

Такая команда import сообщает компилятору TypeScript, что местоположение модуля задается относительно файла, содержащего команду import. В данном случае файл NameAndWeather.ts находится в каталоге modules, в котором находится и файл primer.ts. Во втором варианте импортирования местоположение задается абсолютно. Ниже приведен пример абсолютного импортирования из главы 2; этот способ встречается практически в каждой главе этой книги:

```
...
import { Component } from "@angular/core";
...
```

Команда import не начинается с ./, поэтому компилятор TypeScript разрешает его с использованием пакетов NPM из папки node_modules.

Также необходимо настроить загрузчик модулей JavaScript, чтобы он умел разрешать как относительные, так и абсолютные модули с использованием запросов HTTP. Этот процесс более подробно описан в главе 11.

Назначение псевдонимов при импортировании

В сложных проектах с большим количеством зависимостей может возникнуть ситуация, при которой вам придется использовать два одноименных класса из разных модулей. Чтобы воссоздать эту проблему, я создал файл DuplicateName.ts в папке JavaScriptPrimer/modules и определил класс, показанный в листинге 6.9.

Листинг 6.9. Содержимое файла DuplicateName.ts в папке JavaScriptPrimer/modules

```
export class Name {

    get message() {
        return "Other Name";
    }
}
```

Этот класс не делает ничего полезного, но он называется `Name`; это означает, что при импортировании его методом из листинга 6.8 возникнет конфликт, потому что компилятор не сможет различить классы. Проблема решается при помощи

ключевого слова as, которое назначает псевдоним для класса при импортировании его из модуля (листинг 6.10).

Листинг 6.10. Использование псевдонима при импортировании в файле primer.ts

```
import { Name, WeatherLocation } from "./modules/NameAndWeather";
import { Name as OtherName } from "./modules/DuplicateName";

let name = new Name("Adam", "Freeman");
let loc = new WeatherLocation("raining", "London");
let other = new OtherName();

console.log(name.nameMessage);
console.log(loc.weatherMessage);
console.log(other.message);
```

Класс Name из модуля DuplicateName импортируется под псевдонимом OtherName, что позволяет использовать его без конфликтов с классом Name в модуле NameAndWeather. Пример выдает следующий результат:

```
Hello Adam Freeman

It is raining in London
Other Name
```

Импортирование всех типов в модуле

Альтернативное решение заключается в импортировании модуля в виде объекта, который обладает свойствами для всех содержащихся в нем типов (листинг 6.11).

Листинг 6.11. Импортирование модуля как объекта в файле primer.ts

```
import * as NameAndWeatherLocation from "./modules/NameAndWeather";
import { Name as OtherName } from "./modules/DuplicateName";

let name = new NameAndWeatherLocation.Name("Adam", "Freeman");
let loc = new NameAndWeatherLocation.WeatherLocation("raining", "London");
let other = new OtherName();

console.log(name.nameMessage);
console.log(loc.weatherMessage);
console.log(other.message);
```

Команда import в этом примере импортирует содержимое модуля NameAndWeather и создает объект с именем NameAndWeatherLocation. Этот объект содержит свойства Name и Weather, соответствующие классам, определенным в модуле. Пример выдает тот же результат, что и листинг 6.10.

ПРИМЕЧАНИЕ

Не все модули экспортируют типы; это означает, что вы не сможете задать импортируемые имена. В такой ситуации можно воспользоваться командой import, в которой указывается имя файла JavaScript без расширения. Такой способ нередко применяется на практике, поэтому вы встретите такие команды, как import “rxjs/add/operator/map” из

приложения SportsStore главы 8. Эта команда заставляет загрузчик модулей выполнить содержимое файла rxjs/add/operator/map.js, который добавляет метод для обработки данных, полученных из запроса HTTP.

Полезные возможности TypeScript

TypeScript является надмножеством JavaScript; таким образом, это не просто компилятор для работы с новейшими возможностями JavaScript. Ниже я продемонстрирую самые полезные возможности TypeScript для разработки Angular, многие из которых используются в примерах книги.

Возможности, описанные в этом разделе, не являются строго необходимыми. В разработке приложений Angular вполне можно обойтись и без них; ничто не мешает вам создавать сложные, полнофункциональные приложения только на базе возможностей JavaScript, описанных в главе 5 и предыдущих разделах этой главы. Однако эти возможности TypeScript упрощают использование некоторых средств Angular, как будет показано в следующих главах.

ПРИМЕЧАНИЕ

В этой главе описана лишь часть возможностей TypeScript. Другие возможности будут описаны при их использовании в последующих главах, а полную информацию можно получить на домашней странице TypeScript по адресу www.typescriptlang.org.

Аннотации типов

Гибкость системы типов JavaScript делит разработчиков на два лагеря. Одни считают типы JavaScript мощными, выразительными и динамичными. Для других они порождают хаос до такой степени, что у многих остается впечатление, что в JavaScript системы типов нет вообще.

Одна из ключевых возможностей TypeScript — поддержка аннотаций типов, которые помогают предотвратить типичные ошибки JavaScript за счет проверки типов во время компиляции (по аналогии с тем, как это делается в языках C# или Java). Если вы так и не примирились с системой типов JavaScript (или даже не поняли, что она существует), аннотации типов сильно снижают риск типичных ошибок. (С другой стороны, любители полной свободы могут посчитать, что ограничения, связанные с аннотациями типов TypeScript, только мешают и раздражают.)

Чтобы показать, какого рода задачи решаются с использованием аннотаций типов, я создал файл tempConverter.ts в папке JavaScriptPrimer и добавил код из листинга 6.12.

Листинг 6.12. Содержимое файла tempConverter.ts в папке JavaScriptPrimer

```
export class TempConverter {

    static convertFtoC(temp) {
        return ((parseFloat(temp.toPrecision(2)) - 32) / 1.8).toFixed(1);
    }
}
```

Класс `TempConverter` содержит простой статический метод `convertFtoC`, который получает температуру по шкале Фаренгейта и возвращает ту же температуру по шкале Цельсия.

Проблема в том, что код содержит ряд неявно выраженных предположений. Предполагается, что метод `convertFtoC` получает значение `number`, для которого вызывается метод `toPrecision` для ограничения цифр в дробной части. Метод возвращает строку, хотя это и не столь очевидно без тщательного анализа кода (результатом метода `toFixed` является `string`).

Все эти неявные предположения порождают проблемы, особенно когда один разработчик использует код JavaScript, написанный другим разработчиком. В листинге 6.13 я намеренно внес ошибку, передавая температуру в виде значения `string` (вместо значения `number`, которое ожидает получить метод).

Листинг 6.13. Передача неправильного типа в файле primer.ts

```
import { Name, WeatherLocation } from "./modules/NameAndWeather";
import { Name as OtherName } from "./modules/DuplicateName";
import { TempConverter } from "./tempConverter";

let name = new Name("Adam", "Freeman");
let loc = new WeatherLocation("raining", "London");
let other = new OtherName();

let cTemp = TempConverter.convertFtoC("38");

console.log(name.nameMessage);
console.log(loc.weatherMessage);
console.log(`The temp is ${cTemp}C`);
```

При выполнении этого кода в браузере на консоль JavaScript будет выведено следующее сообщение (точный вид сообщения может отличаться в зависимости от используемого браузера):

```
temp.toPrecision is not a function
```

Конечно, подобные проблемы могут решаться и без использования JavaScript, но это означает, что значительная часть кода любого приложения JavaScript будет заниматься проверкой используемых типов. В решении на базе TypeScript эта работа поручается компилятору: в код JavaScript добавляются аннотации типов. В листинге 6.14 я добавил аннотации типов в класс `TempConverter`.

Листинг 6.14. Добавление аннотаций типов в файл tempConverter.ts

```
export class TempConverter {

    static convertFtoC(temp: number) : string {
        return ((parseFloat(temp.toPrecision(2)) - 32) / 1.8).toFixed(1);
    }
}
```

Аннотации типов состоят из двоеточия (`:`) и типа. В этом примере используются две аннотации. Первая сообщает, что параметр метода `convertFtoC` должен быть числом (`number`).

```
...
static convertFtoC(temp: number) : string {
...
```

Другая аннотация сообщает, что результат метода должен быть строкой (`string`).

```
...
static convertFtoC(temp: number) : string {
...
```

Когда вы сохраните изменения в файле, запускается компилятор TypeScript. Среди полученных сообщений об ошибках имеется следующее:

```
primer.ts(8,39): error TS2345: Argument of type 'string' is not assignable to
parameter of type 'number'.
```

Компилятор TypeScript обнаружил, что тип значения, переданного методу `convertFtoC` в файле primer.ts, не соответствует аннотации типа, и сообщил об ошибке. В этом заключается суть системы типов TypeScript: вам не придется писать в своих классах дополнительный код для проверки того, что были получены значения ожидаемых типов. Кроме того, компилятору будет проще определить тип результата метода. Чтобы избавиться от ошибки, о которой сообщает компилятор, в листинге 6.15 команда вызова `convertFtoC` изменяется так, чтобы при вызове передавалось число.

Листинг 6.15. Использование числового аргумента в файле primer.ts

```
import { Name, WeatherLocation } from "./modules/NameAndWeather";
import { Name as OtherName } from "./modules/DuplicateName";
import { TempConverter } from "./tempConverter";

let name = new Name("Adam", "Freeman");
let loc = new WeatherLocation("raining", "London");
let other = new OtherName();

let cTemp = TempConverter.convertFtoC(38);

console.log(name.nameMessage);
console.log(loc.weatherMessage);
console.log(other.message);
console.log(`The temp is ${cTemp}C`);
```

При сохранении изменений на консоль JavaScript в браузере выводятся следующие сообщения:

```
Hello Adam Freeman
It is raining in London
Other Name
The temp is 3.3C
```

Применение аннотаций типов к свойствам и переменным

Аннотации типов применяются не только к методам. Они также могут применяться к свойствам и переменным, что гарантирует, что все типы, используемые в приложении, могут проверяться компилятором. В листинге 6.16 аннотации типов были добавлены в классы из модуля `NameAndWeather`.

ОПРЕДЕЛЕНИЕ ТИПОВ ДЛЯ БИБЛИОТЕК JAVASCRIPT

TypeScript лучше всего работает при наличии полной информации о типах для всех пакетов, с которыми вы работаете, включая сторонние пакеты. Некоторые пакеты, необходимые для Angular (включая сам фреймворк Angular), включают информацию, необходимую для TypeScript. Для других пакетов информацию о типах требуется загрузить и добавить в пакет.

Для этого используется программа typings, которая используется в главе 7; она более подробно рассматривается в главе 11.

Листинг 6.16. Применение аннотаций типов в файле NameAndWeather.ts

```
export class Name {
    first: string;
    second: string;

    constructor(first: string, second: string) {
        this.first = first;
        this.second = second;
    }

    get nameMessage() : string {
        return `Hello ${this.first} ${this.second}`;
    }
}

export class WeatherLocation {
    weather: string;
    city: string;

    constructor(weather: string, city: string) {
        this.weather = weather;
        this.city = city;
    }

    get weatherMessage() : string {
        return `It is ${this.weather} in ${this.city}`;
    }
}
```

Свойства объявляются с аннотацией типа по той же схеме, что и параметры и результаты функций. Изменения в листинге 6.17 устраняют остальные ошибки, о которых сообщает компилятор TypeScript, — проблемы были обусловлены тем, что компилятор не знал тип свойств, создаваемых в конструкторах.

Схема с передачей параметров конструктору и присваиванием их значений переменным встречается настолько часто, что в TypeScript для нее предусмотрена специальная оптимизация (листинг 6.17).

Листинг 6.17. Создание свойств на основе параметров конструктора в файле NameAndWeather.ts

```
export class Name {
    constructor(private first: string, private second: string) {}

    get nameMessage() : string {
        return `Hello ${this.first} ${this.second}`;
    }
}

export class WeatherLocation {

    constructor(private weather: string, private city: string) {}

    get weatherMessage() : string {
        return `It is ${this.weather} in ${this.city}`;
    }
}
```

Ключевое слово `private` — пример модификатора управления доступом (см. раздел «Модификаторы доступа»). Применение ключевого слова к параметру конструктора приводит к автоматическому определению свойства класса и присваивания ему значения параметра. Код в листинге 6.17 работает так же, как листинг 6.16, но записывается более компактно.

Определение множественных типов или произвольного типа

TypeScript позволяет задать сразу несколько типов, разделяемых вертикальной чертой (символ |). Такая возможность полезна, если метод может получать или возвращать разные типы или когда переменной могут присваиваться значения разных типов. В листинге 6.18 метод `convertFtoC` изменяется таким образом, чтобы он мог получать значения `number` или `string`.

Листинг 6.18. Получение значений разных типов в файле tempConverter.ts

```
export class TempConverter {

    static convertFtoC(temp: number | string): string {
        let value: number = (<number>temp).toPrecision
            ? <number>temp : parseFloat(<string>temp);
        return ((parseFloat(value.toPrecision(2)) - 32) / 1.8).toFixed(1);
    }
}
```

Объявление типа параметра `temp` преобразовано к виду `number | string`; это означает, что метод может получить любое значение. Такая конструкция называется *объединенным типом*. Внутри метода мы определяем, какой именно тип был получен. Конструкция получается немного громоздкой, но значение параметра пре-

образуется к значению number для проверки того, что для результата определен метод toPrecision:

```
...
(<number>temp).toPrecision
...
```

Угловые скобки (< и >) преобразуют объект к заданному типу. Того же результата можно добиться при помощи ключевого слова as (листинг 6.19).

Листинг 6.19. Ключевое слово as в файле tempConverter.ts

```
export class TempConverter {

    static convertFtoC(temp: number | string): string {
        let value: number = (temp as number).toPrecision
            ? temp as number : parseFloat(<string>temp);
        return ((parseFloat(value.toPrecision(2)) - 32) / 1.8).toFixed(1);
    }
}
```

Вместо задания объединенного типа также можно воспользоваться ключевым словом any, которое позволяет присвоить любой тип переменной, использовать его в аргументе или вернуть из метода. В листинге 6.20 объединенный тип в методе convertFtoC заменяется ключевым словом any.

ПРИМЕЧАНИЕ

При отсутствии аннотации типа компилятор TypeScript неявно применяет ключевое слово any.

Листинг 6.20. Использование типа any в файле tempConverter.ts

```
export class TempConverter {

    static convertFtoC(temp: any): string {
        let value: number;
        if ((temp as number).toPrecision) {
            value = temp;
        } else if ((temp as string).indexOf) {
            value = parseFloat(<string>temp);
        } else {
            value = 0;
        }
        return ((parseFloat(value.toPrecision(2)) - 32) / 1.8).toFixed(1);
    }
}
```

ПРИМЕЧАНИЕ

На другом конце спектра находится ключевое слово void, которое сообщает, что метод не возвращает результат. Использовать void не обязательно, но оно явно и недвусмысленно сообщает об отсутствии результата.

Кортежи

Кортежи (tuples) представляют собой массивы фиксированной длины; каждый элемент массива относится к указанному типу. Описание звучит несколько туманно, что объясняется исключительной гибкостью кортежей. Например, в листинге 6.21 кортеж используется для представления города, текущей погоды и температуры.

Листинг 6.21. Использование кортежа в файле primer.ts

```
import { Name, WeatherLocation } from "./modules/NameAndWeather";
import { Name as OtherName } from "./modules/DuplicateName";
import { TempConverter } from "./tempConverter";

let name = new Name("Adam", "Freeman");
let loc = new WeatherLocation("raining", "London");
let other = new OtherName();

let cTemp = TempConverter.convertFtoC("38");

let tuple: [string, string, string];
tuple = ["London", "raining", TempConverter.convertFtoC("38")]

console.log(`It is ${tuple[2]} degrees C and ${tuple[1]} in ${tuple[0]}`);
```

Кортежи определяются как массивы типов, а для обращения к отдельным элементам используются индексы массива. Этот пример выводит следующее сообщение на консоль JavaScript в браузере:

```
It is 3.3 degrees C and raining in London
```

Индексируемые типы

Индексируемые типы связывают ключ со значением; так создается коллекция — разновидность ассоциативного массива, которая может использоваться для группировки взаимосвязанных данных. В листинге 6.22 индексируемый тип используется для группировки информации о нескольких городах.

Листинг 6.22. Использование индексируемого типа в файле primer.ts

```
import { Name, WeatherLocation } from "./modules/NameAndWeather";
import { Name as OtherName } from "./modules/DuplicateName";
import { TempConverter } from "./tempConverter";

let cities: { [index: string]: [string, string] } = {};

cities["London"] = ["raining", TempConverter.convertFtoC("38")];
cities["Paris"] = ["sunny", TempConverter.convertFtoC("52")];

cities["Berlin"] = ["snowing", TempConverter.convertFtoC("23")];

for (let key in cities) {
    console.log(`${key}: ${cities[key][0]}, ${cities[key][1]}`);
}
```

Переменная `cities` определяется как индексируемый тип со строковым ключом и значением данных в виде кортежа `[string, string]`. Для записи и чтения значений используются индексаторы, похожие на индексаторы массивов: `cities["London"]`. Для обращения к коллекции ключей индексируемого типа используется цикл `for...in`, как в этом примере, который выводит следующий результат на консоль JavaScript в браузере:

```
London: raining, 3.3
Paris: sunny, 11.1
Berlin: snowing, -5.0
```

Ключами индексируемых типов могут быть только значения `number` и `string`. Тем не менее это полезная возможность, которая будет использоваться в примерах следующих глав.

Модификаторы доступа

JavaScript не поддерживает ограничения доступа; это означает, что к классам, их свойствам и методам можно обращаться из любой части приложения. Разработчики обычно снабжают внутренние имена префиксом _, но это всего лишь предупреждение для других разработчиков, которое может быть нарушено.

TypeScript предоставляет три ключевых слова для управления доступом, и их соблюдение отслеживается компилятором. Эти ключевые слова перечислены в табл. 6.2.

ПРИМЕЧАНИЕ

В ходе разработки эффект этих ключевых слов в приложениях Angular ограничен, потому что большая часть функциональности реализуется через свойства и методы, которые используются во фрагментах кода, встроенных в выражения привязки данных. Эти выражения обрабатываются во время выполнения в браузере, где никакие ограничения TypeScript не соблюдаются. Они начнут играть более важную роль, когда вы перейдете к развертыванию своего приложения. Обязательно проследите за тем, чтобы все свойства и методы, используемые в выражениях привязки данных, помечались как открытые (public) или не имели модификатора доступа (что равнозначно использованию ключевого слова public).

Таблица 6.2. Ключевые слова модификаторов доступа TypeScript

Ключевое слово	Описание
public	Ключевое слово используется для обозначения свойств или методов, доступных в любой точке. Этот уровень доступа используется по умолчанию, если уровень доступа не задан явно
private	Ключевое слово используется для обозначения свойств или методов, доступных только в пределах класса, в котором они определяются
protected	Ключевое слово используется для обозначения свойств или методов, доступных только в пределах класса, в котором они определяются, или классов, расширяющих этот класс

В листинге 6.23 в класс `TempConverter` добавляется метод с модификатором `private`.

Листинг 6.23. Использование модификатора доступа в файле tempConverter.ts

```
export class TempConverter {
    static convertFtoC(temp: any): string {
        let value: number;
        if ((temp as number).toPrecision) {
            value = temp;
        } else if ((temp as string).indexOf) {
            value = parseFloat(<string>temp);
        } else {
            value = 0;
        }
        return TempConverter.performCalculation(value).toFixed(1);
    }

    private static performCalculation(value: number): number {
        return (parseFloat(value.toPrecision(2)) - 32) / 1.8;
    }
}
```

Метод `performCalculation` помечается с уровнем доступа `private`; это означает, что компилятор TypeScript выдаст сообщение об ошибке, если любая другая часть приложения попытается вызвать этот метод.

Итоги

В этой главе описан механизм поддержки работы с объектами и классами в JavaScript, логика работы с модулями в JavaScript, а также средства TypeScript, которые могут пригодиться при разработке Angular. В следующей главе мы начнем строить реалистичный проект, в ходе которого вы получите представление о том, как разные части Angular взаимодействуют при создании приложений.

7 SportsStore: реальное приложение

В главе 2 мы построили простое приложение Angular. Небольшие целенаправленные примеры позволяют продемонстрировать конкретные возможности Angular, но у них не хватает контекста. Чтобы преодолеть эту проблему, мы создадим простое, но довольно реалистичное приложение из области электронной коммерции.

В нашем приложении, которое называется SportsStore, будет использован классический подход, реализуемый в интернет-магазинах по всему миру. Мы создадим электронный каталог товаров, который клиенты могут просматривать по категориям и по страницам; корзину, в которую пользователи могут добавлять или удалять товары; и процедуру оформления заказа, в которой клиенты смогут вводить данные для доставки и размещать заказы. Также будет создана административная область, которая включает операции создания, чтения, обновления и удаления (CRUD) для управления каталогом, и организована система защиты, чтобы изменения могли вноситься только администраторами с подтвержденными полномочиями. Наконец, я покажу, как подготовить и развернуть приложение Angular.

В этой и следующих главах я постараюсь дать некоторое представление о том, как проходит процесс разработки Angular, на реалистичном примере. Разумеется, при этом я хочу сосредоточиться на Angular, поэтому аспекты интеграции с внешними системами (например, базами данных) по возможности упрощены, а другие (например, обработка платежей) полностью опущены.

Я использую пример SportsStore в нескольких своих книгах — отчасти потому, что он демонстрирует возможность использования разных фреймворков, языков и стилей разработки для достижения того же результата. Вам не придется читать другие книги, чтобы понять материал этой главы, но если у вас уже есть другая книга (например, «Pro ASP.NET Core MVC»), сравнение может быть весьма интересным.

Функциональность Angular, которую я использую в приложении SportsStore, подробно рассматривается в следующих главах. Вместо того чтобы повторять здесь весь материал, я расскажу достаточно для того, чтобы вы поняли логику примера, и дам ссылки на другие главы, в которых можно получить подробную информацию. Поэтому прочитайте главы, посвященные SportsStore, и получите представление о том, как работает Angular, либо сразу переходите к главам с подробной информацией для углубленного изучения темы. В любом случае не ожидайте, что все будет понятно с самого начала, — в Angular много взаимодействующих компонентов. Приложение SportsStore должно показать, как они взаимодействуют, не отвлекаясь на подробное изложение материала, который будет рассматриваться в оставшейся части книги.

Подготовка проекта

Чтобы создать проект SportsStore, откройте командную строку, перейдите в удобное место и выполните следующую команду:

```
ng new SportsStore
```

Пакет `angular-cli` создает новый проект для разработки Angular с файлами конфигурации, временным контентом и другими средствами разработчика. Процесс подготовки проекта может занять некоторое время из-за загрузки и установки большого количества пакетов NPM.

Создание структуры папок

Отправной точкой для любого приложения Angular становится создание структуры папок. Команда `ng new` создает структуру, в которой все файлы приложения находятся в папке src, а файлы Angular — в папке src/app. Чтобы определить дополнительную структуру проекта, создайте папки, перечисленные в табл. 7.1.

Таблица 7.1. Дополнительные папки, необходимые для проекта SportsStore

Папка	Описание
SportsStore/src/app/model	Папка для кода модели данных
SportsStore/src/app/store	Папка для базовой функциональности совершения покупок
SportsStore/src/app/admin	Папка для функциональности администрирования

Установка дополнительных пакетов NPM

Для проекта SportsStore нужны некоторые дополнительные пакеты кроме базовых пакетов, настраиваемых `angular-cli`. Отредактируйте файл package.json в папке SportsStore и внесите в него дополнения, представленные в листинге 7.1.

ВНИМАНИЕ

Чтобы ваши результаты соответствовали результатам примеров в этой и других главах, очень важно использовать во всех примерах книги конкретные версии, указанные в листинге. Если у вас возникнут проблемы с примерами этой или какой-либо из последующих глав, попробуйте использовать исходный код из архива, прилагаемого к книге; он содержит дополнительную конфигурационную информацию для NPM с указанием версий каждого пакета и его зависимостей. Если же положение окажется совсем безвыходным, отправьте мне сообщение по адресу adam@adam-freeman.com, я постараюсь помочь вам.

Листинг 7.1. Содержимое файла package.json из папки SportsStore

```
{
  "name": "sports-store",
  "version": "0.0.0",
  "license": "MIT",
```

```
  "scripts": {
    "ng": "ng",
    "start": "ng serve",
    "build": "ng build",
    "test": "ng test",
    "lint": "ng lint",
    "e2e": "ng e2e",
    "json": "json-server data.js -p 3500 -m authMiddleware.js"
  },
  "private": true,
  "dependencies": {
    "@angular/common": "^4.0.0",
    "@angular/compiler": "^4.0.0",
    "@angular/core": "^4.0.0",
    "@angular/forms": "^4.0.0",
    "@angular/http": "^4.0.0",
    "@angular/platform-browser": "^4.0.0",
    "@angular/platform-browser-dynamic": "^4.0.0",
    "@angular/router": "^4.0.0",
    "core-js": "^2.4.1",
    "rxjs": "^5.1.0",
    "zone.js": "^0.8.4",
    "bootstrap": "4.0.0-alpha.4",
    "font-awesome": "4.7.0"
  },

  "devDependencies": {
    "@angular/cli": "1.0.0",
    "@angular/compiler-cli": "^4.0.0",
    "@types/jasmine": "2.5.38",
    "@types/node": "~6.0.60",
    "codelyzer": "~2.0.0",
    "jasmine-core": "~2.5.2",
    "jasmine-spec-reporter": "~3.2.0",
    "karma": "~1.4.1",
    "karma-chrome-launcher": "~2.0.0",
    "karma-cli": "~1.0.1",
    "karma-jasmine": "~1.1.0",
    "karma-jasmine-html-reporter": "^0.2.2",
    "karma-coverage-istanbul-reporter": "^0.2.0",
    "protractor": "~5.1.0",
    "ts-node": "~2.0.0",
    "tslint": "~4.5.0",
    "typescript": "~2.2.0",
    "json-server": "0.8.21",
    "jsonwebtoken": "7.1.9"
  }
}
```

Сохраните файл package.json и выполните следующую команду из папки SportsStore, чтобы загрузить и установить пакеты, необходимые для проекта:

```
npm install
```

NPM выводит полный список пакетов после того, как они будут установлены. Обычно в процессе установки выводятся некритические предупреждения, на которые можно не обращать внимания.

Подготовка REST-совместимой веб-службы

Приложение SportsStore использует асинхронные запросы HTTP для получения модели данных, предоставляемой REST-совместимой веб-службой. Как будет показано в главе 24, REST — стиль проектирования веб-служб, использующий методы (или команды) HTTP для определения операций и URL-адреса для выбора объектов данных, к которым применяется операция.

Я включил в файл package.json строку `json-server`; это отличный пакет для быстрого создания веб-служб на основе данных JSON или кода JavaScript. Чтобы определить фиксированное состояние, к которому можно вернуть проект, я воспользуюсь возможностью передачи данных REST-совместимой веб-службе из кода JavaScript; это означает, что перезапуск веб-службы приведет к сбросу данных приложения. Создайте файл data.js в папке SportsStore и включите в него код из листинга 7.2.

ПРИМЕЧАНИЕ

При создании конфигурационных файлов обращайте внимание на имена файлов. Некоторые файлы имеют расширение .json; это означает, что они содержат статические данные в формате JSON. Другие файлы с расширением .js содержат код JavaScript; каждый инструмент, необходимый для разработки Angular, предъявляет некоторые требования к конфигурационному файлу.

Листинг 7.2. Содержимое файла data.js в папке SportsStore

```
module.exports = function () {
    return {
        products: [
            { id: 1, name: "Kayak", category: "Watersports",
                description: "A boat for one person", price: 275 },
            { id: 2, name: "Lifejacket", category: "Watersports",
                description: "Protective and fashionable", price: 48.95 },
            { id: 3, name: "Soccer Ball", category: "Soccer",
                description: "FIFA-approved size and weight", price: 19.50 },
            { id: 4, name: "Corner Flags", category: "Soccer",
                description: "Give your playing field a professional touch",
                price: 34.95 },
            { id: 5, name: "Stadium", category: "Soccer",
                description: "Flat-packed 35,000-seat stadium", price: 79500 },
            { id: 6, name: "Thinking Cap", category: "Chess",
                description: "Improve brain efficiency by 75%", price: 16 },
            { id: 7, name: "Unsteady Chair", category: "Chess",
                description: "Secretly give your opponent a disadvantage",
                price: 29.95 },
            { id: 8, name: "Human Chess Board", category: "Chess",
                description: "A fun game for the family", price: 75 },
            { id: 9, name: "Bling Bling King", category: "Chess",
```

```
                description: "Gold-plated, diamond-studded King", price: 1200 }
        ],
        orders: []
    }
}
```

Этот код определяет две коллекции данных, которые будут представляться REST-совместимой веб-службой. Коллекция `products` содержит товары, которые продаются в интернет-магазине, а коллекция `orders` содержит заказы, размещенные пользователями (в настоящее время она пуста).

Данные, хранимые REST-совместимой веб-службой, необходимо защитить, чтобы обычные пользователи не могли изменять описания товаров или состояние своих заказов. Пакет `json-server` не содержит встроенных средств аутентификации, поэтому я создал файл с именем authMiddleware.js в папке SportsStore и добавил код из листинга 7.3.

Листинг 7.3. Содержимое файла authMiddleware.js в папке SportsStore

```
const jwt = require("jsonwebtoken");

const APP_SECRET = "myappsecret";
const USERNAME = "admin";
const PASSWORD = "secret";

module.exports = function (req, res, next) {
    if (req.url == "/login" && req.method == "POST") {
        if (req.body != null && req.body.name == USERNAME
                && req.body.password == PASSWORD) {
            let token = jwt.sign({ data: USERNAME, expiresIn: "1h" }, APP_SECRET);
            res.json({ success: true, token: token });
        } else {
            res.json({ success: false });
        }
        res.end();
        return;
    } else if ((req.url.startsWith("/products") && req.method != "GET")
                || (req.url.startsWith("/orders") && req.method != "POST")) {
         let token = req.headers["authorization"];
         if (token != null && token.startsWith("Bearer<")) {
            token = token.substring(7, token.length - 1);
            try {
                jwt.verify(token, APP_SECRET);
                next();
                return;
            } catch (err) {}
         }
        res.statusCode = 401;
        res.end();
        return;
    }
    next();
}
```

Этот код проверяет запросы HTTP, отправленные REST-совместимой веб-службе, и реализует простейшие средства безопасности. Это серверный код, не связанный напрямую с разработкой приложений Angular, поэтому если его смысл неочевиден — не беспокойтесь. Процессы аутентификации и авторизации, включая механизм аутентификации пользователей в Angular, будут более подробно рассмотрены в главе 9.

ВНИМАНИЕ

Не используйте код из листинга 7.3 в других приложениях. Он содержит слабые пароли, жестко зафиксированные в коде. В проекте SportsStore это нормально, потому что нас прежде всего интересует разработка на стороне клиента с применением Angular, но для реальных проектов такие решения не подходят.

Подготовка файла HTML

У каждого веб-приложения Angular имеется файл HTML, который загружается браузером и выполняет загрузку и запуск приложения. Отредактируйте файл index.html в папке SportsStore/src и добавьте в него элементы из листинга 7.4.

Листинг 7.4. Содержимое файла index.html в папке SportsStore/src

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>SportsStore</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link href="node_modules/bootstrap/dist/css/bootstrap.min.css"
        rel="stylesheet" />
</head>
<body class="m-a-1">
    <app>SportsStore Will Go Here</app>
</body>
</html>
```

Документ HTML включает элементы `script` для библиотек полизаполнений, необходимых для поддержки старых браузеров, элемент `link` для загрузки таблицы стилей Bootstrap и элемент `app`, который резервирует место для функциональности SportsStore.

ПРИМЕЧАНИЕ

Документ HTML включает элемент base, необходимый для работы средств маршрутизации URL в Angular. Эта функциональность будет добавлена в проект SportsStore в главе 8.

Запуск примера

Убедитесь в том, что все файлы были сохранены, и выполните следующую команду из папки SportsStore:

```
ng serve --port 3000 --open
```

Команда запускает цепочку инструментов разработки, настроенную `angular-cli`, которая автоматически компилирует и упаковывает код и файлы контента из папки src при обнаружении любых изменений. Открывается новое окно браузера с контентом, представленным на рис. 7.1.

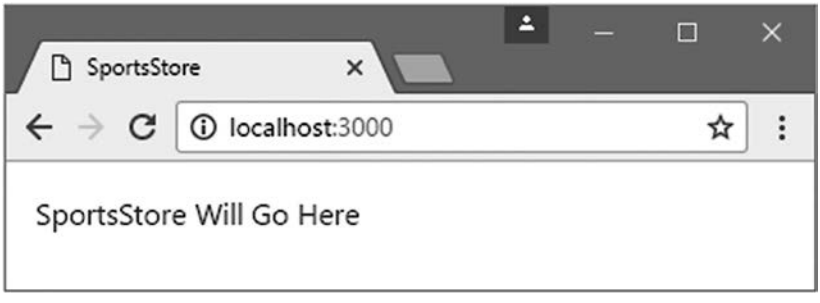


Рис. 7.1. Запуск приложения

Веб-сервер для разработки работает на порте 3000, поэтому URL-адрес приложения имеет вид *http://localhost:3000*. Включать имя документа HTML не обязательно, потому что имя index.html используется по умолчанию сервером для ответа.

Запуск REST-совместимой веб-службы

Чтобы запустить REST-совместимую веб-службу, откройте новое окно командной строки, перейдите в папку SportsStore и выполните следующую команду:

```
npm run json
```

REST-совместимая веб-служба настроена для работы на порте 3500. Чтобы протестировать запрос к веб-службе, введите в браузере URL *http://localhost:3500/products/1*. Браузер выводит представление одного из товаров, определенных в листинге 7.2, в формате JSON:

```
{
  "id": 1,
  "name": "Kayak",
  "category": "Watersports",
  "description": "A boat for one person",
  "price": 275
}
```

Подготовка проекта Angular

Каждый проект Angular требует определенной базовой подготовки — просто для перехода в состояние, в котором приложение загружается и запускается браузером. В последующих разделах мы заложим основу, на которой будет строиться приложение SportsStore.

Обновление корневого компонента

Начнем с корневого компонента — структурного блока Angular, который будет управлять элементом app в документе HTML. Приложение может содержать несколько компонентов, но среди них всегда присутствует корневой компонент, отвечающий за отображение контента верхнего уровня. Отредактируйте файл app.component.ts в папке SportsStore/src/app и включите в него код из листинга 7.5.

Листинг 7.5. Содержимое файла app.component.ts в папке SportsStore/src/app

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

@Component({
    selector: "app",
    template: `<div class="bg-success p-a-1 text-xs-center">
                    This is SportsStore
               </div>`
})
export class AppComponent { }
```

Декоратор `@Component` сообщает Angular, что класс `AppComponent` является компонентом, а его свойства описывают применение этого компонента. Полный набор свойств компонентов приведен в главе 17, но три свойства, приведенные в листинге, являются основными и часто применяемыми на практике. Свойство `selector` сообщает Angular, как следует применять компонент в документе HTML, а свойство `template` определяет контент, который будет отображаться компонентом. Компоненты могут определять встроенные шаблоны, как в данном случае, или использовать внешние файлы HTML, которые упрощают управление сложным контентом.

Класс `AppComponent` не содержит кода, потому что корневой компонент в проекте Angular существует только для управления контентом, отображаемым для пользователя. На начальной стадии мы будем управлять контентом, отображаемым корневым компонентом, вручную, но в главе 8 будет представлен механизм *маршрутизации URL* для автоматической адаптации контента в зависимости от действий пользователя.

Обновление корневого модуля

Модули Angular делятся на две категории: *функциональные модули* и *корневой модуль*. Функциональные модули используются для группировки взаимосвязанной функциональности приложения, чтобы упростить управление приложением. Мы создадим функциональные модули для всех основных функциональных областей приложения, включая модель данных, пользовательский интерфейс магазина и интерфейс администрирования.

Корневой модуль передает описание приложения для Angular. В описании указано, какие функциональные модули необходимы для запуска приложения, какие нестандартные возможности следует загрузить и как называется корневой компонент. Традиционно файлу корневого компонента присваивается имя app.module.ts;

создайте файл с таким именем в папке SportsStore/src/app и включите код из листинга 7.6.

Листинг 7.6. Содержимое файла app.module.ts в папке SportsStore/src/app

```
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 { }
```

По аналогии с корневым компонентом класс корневого модуля не содержит код. Дело в том, что корневой модуль существует только для передачи информации через декоратор `@NgModule`. Свойство `imports` приказывает Angular загрузить функциональный модуль `BrowserModule` со всей основной функциональностью Angular, необходимой для веб-приложения.

Свойство `declarations` приказывает Angular загрузить корневой компонент, а свойство `bootstrap` сообщает, что корневым компонентом является класс `AppModule`. Информация будет добавлена в свойства этого декоратора при включении функциональности в приложение SportsStore, но для запуска приложения будет достаточно и базовой конфигурации.

Анализ файла начальной загрузки

Следующий блок служебного кода — файл начальной загрузки, запускающий приложение. В книге основное внимание уделяется применению Angular для создания приложений, работающих в браузерах, но платформа Angular может портироваться в разные среды. Файл начальной загрузки использует браузерную платформу Angular для загрузки корневого модуля и запуска приложения. Создайте файл с именем main.ts, традиционно назначаемым файлу начальной загрузки, в папке SportsStore/src/app и добавьте в него код из листинга 7.7.

Листинг 7.7. Содержимое файла main.ts в папке SportsStore/src

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

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) {
  enableProdMode();
}

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

Инструменты разработки обнаруживают изменения в файле проекта, компилируют файлы с кодом и автоматически перезагружают браузер с выводом контента, изображенного на рис. 7.2.

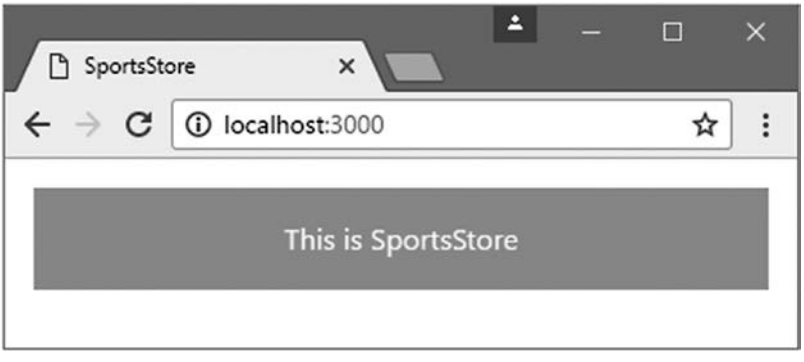


Рис. 7.2. Запуск приложения SportsStore

Просматривая модель DOM в браузере, вы увидите, что временный контент из шаблона корневого компонента был вставлен между начальным и конечным тегами элемента app:

```
<body class="m-a-1">
    <app>
        <div class="bg-success p-a-1 text-xs-center">
            This is SportsStore
        </div>
    </app>
</body>
```

Начало работы над моделью данных

Работу над любым новым проектом лучше всего начинать с модели данных. Я хочу поскорее продемонстрировать некоторые возможности Angular в действии, поэтому вместо определения модели данных от начала до конца мы начнем с реализации базовой функциональности на фиктивных данных. Затем эти данные будут использованы для создания интерфейсной части, а в главе 8 мы вернемся к модели данных и свяжем ее с REST-совместимой веб-службой.

Создание классов модели

Каждой модели данных необходимы классы для описания типов данных, входящих в модель данных. В приложении SportsStore это классы с описанием товаров, продаваемых в интернет-магазине, и заказы, полученные от пользователей.

Для начала работы приложения SportsStore достаточно возможности описания товаров; другие классы моделей будут создаваться для поддержки расширенной

функциональности по мере их реализации. Создайте файл с именем product.model.ts в папке SportsStore/src/app/model и включите код из листинга 7.8.

Листинг 7.8. Содержимое файла product.model.ts из папки SportsStore/src/app/model

```
export class Product {

    constructor(
        public id?: number,
        public name?: string,
        public category?: string,
        public description?: string,
        public price?: number) { }
}
```

Класс `Product` определяет конструктор, который получает свойства `id`, `name`, `category`, `description` и `price`. Эти свойства соответствуют структуре данных, используемых для заполнения REST-совместимой веб-службы в листинге 7.2. Вопросительные знаки (`?`) за именами параметров указывают, что это необязательные параметры, которые могут быть опущены при создании новых объектов с использованием класса `Product`; это может быть удобно при разработке приложений, свойства объектов модели которых заполняются с использованием форм HTML.

Создание фиктивного источника данных

Чтобы подготовить переход от фиктивных данных к реальным, мы будем передавать данные приложению из источника данных. Остальной код приложения не знает, откуда поступили данные, и переход на получение данных из запросов HTTP пройдет прозрачно.

Создайте файл static.datasource.ts в папке SportsStore/src/app/model и включите определение класса из листинга 7.9.

Листинг 7.9. Содержимое файла static.datasource.ts из папки SportsStore/src/app/model

```
import { Injectable } from "@angular/core";
import { Product } from "./product.model";
import { Observable } from "rxjs/Observable";
import "rxjs/add/observable/from";

@Injectable()
export class StaticDataSource {
    private products: Product[] = [
        new Product(1, "Product 1", "Category 1", "Product 1 (Category 1)", 100),
        new Product(2, "Product 2", "Category 1", "Product 2 (Category 1)", 100),
        new Product(3, "Product 3", "Category 1", "Product 3 (Category 1)", 100),
        new Product(4, "Product 4", "Category 1", "Product 4 (Category 1)", 100),
        new Product(5, "Product 5", "Category 1", "Product 5 (Category 1)", 100),
        new Product(6, "Product 6", "Category 2", "Product 6 (Category 2)", 100),
        new Product(7, "Product 7", "Category 2", "Product 7 (Category 2)", 100),
        new Product(8, "Product 8", "Category 2", "Product 8 (Category 2)", 100),
```

```
        new Product(9, "Product 9", "Category 2", "Product 9 (Category 2)", 100),
        new Product(10, "Product 10", "Category 2", "Product 10 (Category 2)", 100),
        new Product(11, "Product 11", "Category 3", "Product 11 (Category 3)", 100),
        new Product(12, "Product 12", "Category 3", "Product 12 (Category 3)", 100),
        new Product(13, "Product 13", "Category 3", "Product 13 (Category 3)", 100),
        new Product(14, "Product 14", "Category 3", "Product 14 (Category 3)", 100),
        new Product(15, "Product 15", "Category 3", "Product 15 (Category 3)", 100),
    ];

    getProducts(): Observable<Product[]> {
        return Observable.from([this.products]);
    }
}
```

Класс `StaticDataSource` определяет метод с именем `getProducts`, который возвращает фиктивные данные. Вызов метода `getProducts` возвращает результат `Observable<Product[]>` — реализацию `Observable` для получения массивов объектов `Product`.

Класс `Observable` предоставляется пакетом Reactive Extensions, который используется Angular для обработки изменений состояния в приложениях. Класс `Observable` будет описан в главе 23, а в этой главе достаточно знать, что объект `Observable` похож на объект JavaScript `Promise`: он представляет асинхронную задачу, которая в будущем должна вернуть результат. Angular раскрывает использование объектов `Observable` для некоторых своих функций, включая работу с запросами HTTP; именно поэтому метод `getProducts` возвращает `Observable<Product[]>` вместо возвращения данных — простого синхронного или с использованием `Promise`.

Декоратор `@Injectable` применяется к классу `StaticDataSource`. Этот декоратор сообщает Angular, что этот класс будет использоваться как служба, что позволяет другим классам обращаться к его функциональности через механизм *внедрения зависимостей*, описанный в главах 19 и 20. Когда приложение начнет обретать форму, мы покажем, как работает служба.

ПРИМЕЧАНИЕ

Обратите внимание: для применения декоратора @Injectable мне пришлось импортировать Injectable из модуля JavaScript @angular/core. Я не стану описывать все классы Angular, импортируемые в примере SportsStore; полная информация будет приведена в главах с описанием соответствующих возможностей.

Создание репозитория модели

Источник данных должен предоставить приложению запрашиваемые данные, но обращение к данным обычно происходит через посредника (*репозиторий*), отвечающего за передачу этих данных отдельным структурным блокам приложения, чтобы подробности получения данных оставались скрытыми. Создайте файл product.repository.ts в папке SportsStore/src/app/model и определите класс из листинга 7.10.

Листинг 7.10. Содержимое файла product.repository.ts из папки SportsStore/app/model

```
import { Injectable } from "@angular/core";
import { Product } from "./product.model";
import { StaticDataSource } from "./static.datasource";

@Injectable()
export class ProductRepository {
    private products: Product[] = [];
    private categories: string[] = [];

    constructor(private dataSource: StaticDataSource) {
        dataSource.getProducts().subscribe(data => {
            this.products = data;
            this.categories = data.map(p => p.category)
                .filter((c, index, array) => array.indexOf(c) == index).sort();
        });
    }

    getProducts(category: string = null): Product[] {
        return this.products
            .filter(p => category == null || category == p.category);
    }

    getProduct(id: number): Product {
        return this.products.find(p => p.id == id);
    }

    getCategories(): string[] {
        return this.categories;
    }
}
```

Когда Angular потребуется создать новый экземпляр репозитория, Angular анализирует класс и видит, что для вызова конструктора `ProductRepository` и создания нового объекта ему нужен объект `StaticDataSource`.

Конструктор репозитория вызывает метод `getProducts` источника данных, после чего использует метод `subscribe` объекта `Observable`, возвращаемого для получения данных товаров. За подробностями о работе объектов `Observable` обращайтесь к главе 23.

ПРОСТЫЕ СТРУКТУРЫ ДАННЫХ

Я использовал массив для хранения модели данных, потому что обычно в приложениях Angular лучшие результаты достигаются с простейшими из возможных структур данных. Angular многократно вычисляет выражения в привязках данных при генерировании контента в элементе HTML, а это означает, что более сложные структуры (такие, как класс Map, реализующий коллекцию «ключ — значение» в JavaScript ES6) должны снова и снова преобразовывать свое содержимое в процессе стабилизации приложения Angular. Таким образом, чем проще структура данных, тем меньше работы потребуется для того, чтобы предоставить Angular необходимые данные.

Другая причина для использования простых структур данных — ограниченная поддержка новых возможностей JavaScript в старых браузерах. Например, в случае класса Map компилятор TypeScript ограничивает возможность использования содержимого Map при использовании компилятора для генерирования кода JavaScript, работающего в старых браузерах.

Как следствие, я буду чаще использовать простые структуры данных (особенно массивы) и перейду к написанию более сложных классов для управления данными в массиве. Пример такого рода встретится вам при расширении функциональности класса репозитория товаров для административного интерфейса в главе 9, когда новым функциям потребуется проводить поиск по массиву для нахождения объектов, с которыми должны выполняться операции. Это неэффективно, но такие операции выполняются реже по сравнению с частотой обработки выражений привязки данных Angular.

Создание функционального модуля

Мы определим функциональную модель Angular, которая позволит легко использовать функциональность модели данных в любой точке приложения. Создайте файл с именем model.module.ts в папке SportsStore/app/model и определите класс, приведенный в листинге 7.11.

ПРИМЕЧАНИЕ

Не беспокойтесь, если имена файлов кажутся похожими. Вы привыкнете к структуре приложений Angular в ходе проработки других глав книги. Вскоре вы начнете с первого взгляда определять, для чего нужен тот или иной файл в проекте Angular.

Листинг 7.11. Содержимое файла model.module.ts из папки SportsStore/src/app/model

```
import { NgModule } from "@angular/core";
import { ProductRepository } from "./product.repository";
import { StaticDataSource } from "./static.datasource";

@NgModule({
    providers: [ProductRepository, StaticDataSource]
})
export class ModelModule { }
```

Декоратор `@NgModule` используется для создания функциональных модулей, а его свойства сообщают Angular о том, как должен использоваться модуль. В данном случае модуль содержит всего одно свойство `providers`, которое сообщает, какие классы должны использоваться в качестве служб для механизма внедрения зависимостей, описанного в главах 19 и 20. Функциональные модули и декоратор `@NgModule` описаны в главе 21.

Создание хранилища

Модель данных готова, и мы можем переходить к построению функциональности магазина: просмотру списка товаров и оформлению заказов. В магазине будет использоваться двухстолбцовый макет, список товаров будет фильтроваться при помощи кнопок категорий, а сами товары будут выводиться в таблице (рис. 7.3).

Рис. 7.3. Базовая структура магазина

В следующих разделах мы используем функциональность Angular и данные модели для создания макета, изображенного на иллюстрации.

Создание компонента магазина и шаблона

По мере изучения Angular вы поймете, что одна задача может решаться по-разному с использованием разных комбинаций возможностей. Я попробую применять разные решения в проекте SportsStore для демонстрации некоторых важных возможностей Angular, но пока предпочтение будет отдаваться простым решениям для того, чтобы проект заработал как можно быстрее.

С учетом всего сказанного отправной точкой для функциональности магазина станет новый компонент — класс, предоставляющий данные и логику для шаблона HTML, содержащего привязки данных для динамического генерирования контента. Создайте файл с именем store.component.ts в папке SportsStore/src/app/store и добавьте определение класса в листинге 7.12.

Листинг 7.12. Содержимое файла store.component.ts из папки SportsStore/src/app/store

```
import { Component } from "@angular/core";
import { Product } from "../model/product.model";
import { ProductRepository } from "../model/product.repository";

@Component({
    selector: "store",
    moduleId: module.id,
    templateUrl: "store.component.html"
})
export class StoreComponent {

    constructor(private repository: ProductRepository) { }

    get products(): Product[] {
        return this.repository.getProducts();
```

```
    }

    get categories(): string[] {
        return this.repository.getCategories();
    }
}
```

К классу `StoreComponent` применяется декоратор `@Component`, который сообщает Angular, что класс является компонентом. Свойства декоратора указывают Angular, как применять компонент к контенту HTML (с использованием элемента с именем `store`) и где находится шаблон компонента (в файле с именем store.component.html).

Класс `StoreComponent` предоставляет логику, которая обеспечивает получение контента шаблона.

Конструктор класса получает объект `ProductRepository` в аргументе, который передается через механизм внедрения зависимостей, описанный в главах 20 и 21. Компонент определяет свойства `products` и `categories`, которые будут использоваться для генерирования контента HTML в шаблоне на основании данных, полученных из репозитория.

Чтобы реализовать шаблон компонента, создайте файл store.component.html в папке SportsStore/src/app/store и добавьте контент HTML в листинге 7.13.

Листинг 7.13. Содержимое файла store.component.html из папки SportsStore/src/app/store

```
<div class="navbar navbar-inverse bg-inverse">
    <a class="navbar-brand">SPORTS STORE</a>
</div>
<div class="col-xs-3 bg-info p-a-1">
    {{categories.length}} Categories
</div>
<div class="col-xs-9 bg-success p-a-1">
    {{products.length}} Products
</div>
```

Начальная версия шаблона проста. Большинство элементов предоставляет структуру для макета магазина и применения некоторых CSS-классов Bootstrap. На данный момент используются только две привязки данных Angular, обозначенных символами `{{` и `}}`. Это привязки *строковой интерполяции*; они приказывают Angular вычислить выражение привязки и вставить результат в элемент. Выражения в этих привязках выводят количество продуктов и категорий, предоставляемых компонентом хранилища.

Создание функционального модуля хранилища

Пока объем функциональности магазина невелик, но даже сейчас потребуется дополнительная работа для связывания ее с оставшейся частью приложения. Чтобы создать функциональный модуль Angular для функциональности магазина,

создайте файл с именем store.module.ts в папке SportsStore/src/app/store и добавьте код из листинга 7.14.

Листинг 7.14. Содержимое файла store.module.ts из папки SportsStore/src/app/store

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule } from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { StoreComponent } from "./store.component";

@NgModule({
    imports: [ModelModule, BrowserModule, FormsModule],
    declarations: [StoreComponent],
    exports: [StoreComponent]
})
export class StoreModule { }
```

Декоратор `@NgModule` настраивает модуль; при этом свойство `imports` используется для передачи Angular информации о том, что модуль магазина зависит от модуля модели, а также модулей `BrowserModule` и `FormsModule`, содержащих стандартные функции Angular для веб-приложений и работы с элементами форм HTML. Декоратор использует свойство `declarations` для передачи Angular информации о классе `StoreComponent`, который (как сообщает свойство `exports`) может использоваться в других частях приложения, — это важно, потому что он будет использоваться корневым модулем.

Обновление корневого компонента и корневого модуля

Применение базовой функциональности магазина и модели требует обновления корневого модуля приложения: он должен импортировать два функциональных модуля, а также обновить шаблон корневого модуля для добавления элемента HTML, к которому будет применяться компонент модуля магазина. В листинге 7.15 представлены изменения в шаблоне корневого компонента.

Листинг 7.15. Добавление компонента в файл app.component.ts

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

@Component({
    selector: "app",
    template: "<store></store>"
})
export class AppComponent { }
```

Элемент `store` заменяет предыдущий контент в шаблоне корневого компонента и соответствует значению свойства `selector` декоратора `@Component` в листинге 7.12. В листинге 7.16 показаны изменения, которые необходимо внести в корневой модуль, чтобы среда Angular загружала функциональный модуль с функциональностью магазина.

Листинг 7.16. Импортирование функциональных модулей в файле app.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { AppComponent } from "./app.component";
import { StoreModule } from "./store/store.module";

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

Когда вы сохраните изменения в корневом модуле, Angular будет обладать всей информацией, необходимой для загрузки приложения и отображения контента из модуля магазина (рис. 7.4).

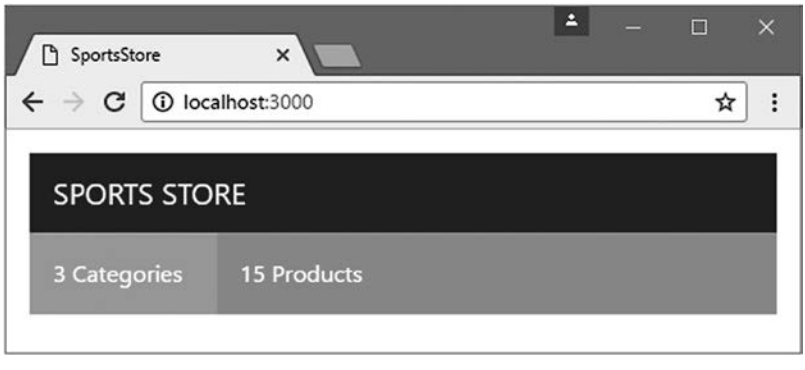


Рис. 7.4. Базовая функциональность приложения SportsStore

Все структурные блоки, созданные в предыдущем разделе, совместно работают для отображения (откровенно говоря, довольно простого) контента, который показывает, сколько в магазине товаров и на сколько категорий они делятся.

Добавление функциональности: подробная информация о товарах

Процесс разработки Angular по своей природе начинается медленно, когда разработчик закладывает фундамент проекта и создает основные структурные блоки. Но когда это будет сделано, новые функции создаются относительно легко. Ниже мы дополним магазин новыми возможностями, чтобы пользователь мог видеть предлагаемые продукты.

Вывод подробной информации о товарах

Очевидная отправная точка для работы над магазином — вывод подробной информации о товарах, чтобы пользователь видел, что же ему предлагают. В лис-

тинге 7.17 в шаблон компонента магазина добавляются элементы HTML с привязками данных, генерирующими контент для каждого товара, предоставляемого компонентом.

Листинг 7.17. Добавление элементов в файл store.component.html

```
<div class="navbar navbar-inverse bg-inverse">
    <a class="navbar-brand">SPORTS STORE</a>
</div>
<div class="col-xs-3 bg-info p-a-1">
    {{categories.length}} Categories
</div>
<div class="col-xs-9 p-a-1">
    <div *ngFor="let product of products" class="card card-outline-primary">
        <h4 class="card-header">
            {{product.name}}
            <span class="pull-xs-right tag tag-pill tag-primary">
                {{ product.price | currency:"USD":true:"2.2-2" }}
            </span>
        </h4>
        <div class="card-text p-a-1">{{product.description}}</div>
    </div>
</div>
```

Большинство элементов управляет макетом и внешним видом контента. Самое важное изменение — добавление выражения привязки данных Angular.

```
...
<div *ngFor="let product of products" class="card card-outline-primary">
...
```

Перед вами пример директивы, трансформирующей элемент HTML, к которому она применяется. Эта конкретная директива `ngFor` преобразует элемент `div`, дублируя его для каждого объекта, возвращаемого свойством `products` компонента. Angular включает ряд встроенных директив для решения большинства типичных задач; эти директивы описаны в главе 13.

При дублировании элемента `div` текущий объект присваивается переменной с именем `product`, что позволяет легко ссылаться на него из других привязок данных — как в следующем примере, где значение свойства `description` текущего товара вставляется как контент элемента `div`:

```
...
<div class="card-text p-a-1">{{product.description}}</div>
...
```

Не все данные в модели данных приложения могут выводиться напрямую для пользователя. Angular включает механизм *каналов* (pipes), используемых классами для преобразования или подготовки значений для привязок данных. Angular содержит несколько встроенных каналов, включая канал `currency`, форматирующий числовые значения в денежном формате:

```
...
{{ product.price | currency:"USD":true:"2.2-2" }}
...
```

Синтаксис применения каналов может показаться немного громоздким, но выражение в этой привязке приказывает Angular отформатировать свойство `price` текущего продукта с использованием канала `currency` по правилам форматирования денежных величин, принятым в США. Сохраните изменения в шаблоне; перечень товаров из модели данных выводится в виде длинного списка (рис. 7.5).

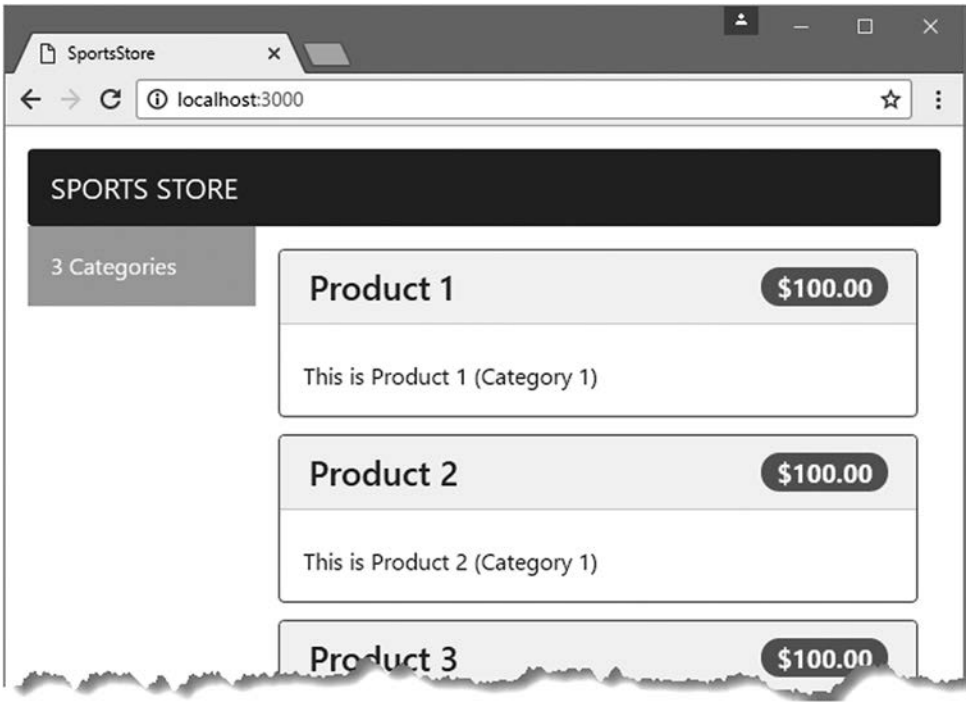


Рис. 7.5. Вывод информации о товаре

Добавление выбора категорий

Чтобы добавить поддержку фильтрации списка товаров по категориям, необходимо подготовить компонент магазина. Он должен следить за тем, какая категория была выбрана пользователем, и изменять механизм выборки данных для использования выбранной категории (листинг 7.18).

Листинг 7.18. Добавление фильтрации по категориям в файл store.component.ts

```
import { Component } from "@angular/core";
import { Product } from "../model/product.model";
import { ProductRepository } from "../model/product.repository";

@Component({
    selector: "store",
    moduleId: module.id,
    templateUrl: "store.component.html"
})
```

```
export class StoreComponent {
    public selectedCategory = null;

    constructor(private repository: ProductRepository) {}

    get products(): Product[] {
        return this.repository.getProducts(this.selectedCategory);
    }

    get categories(): string[] {
        return this.repository.getCategories();
    }

    changeCategory(newCategory?: string) {
        this.selectedCategory = newCategory;
    }
}
```

Изменения просты, потому что они строятся на фундаменте, который мы так долго создавали в начале главы. Свойству `selectedCategory` присваивается выбранная пользователем категория (`null` — все категории); это свойство используется в методе `updateData` как аргумент метода `getProducts`, так что фильтрация делегируется источнику данных. Метод `changeCategory` объединяет эти значения в методе, который может вызываться при выборе категории пользователем.

В листинге 7.19 представлены соответствующие изменения в шаблоне компонента. Шаблон должен отображать набор кнопок для изменения выбранной категории и показывать, какая категория выбрана в настоящее время.

Листинг 7.19. Добавление кнопок категорий в файле store.components.html

```
<div class="navbar navbar-inverse bg-inverse">
    <a class="navbar-brand">SPORTS STORE</a>
</div>
<div class="col-xs-3 p-a-1">
    <button class="btn btn-block btn-outline-primary" (click)="changeCategory()">
        Home
    </button>
    <button *ngFor="let cat of categories" class="btn btn-outline-primary
                    btn-block"
            [class.active]="cat == selectedCategory" (click)="changeCategory(cat)">
        {{cat}}
    </button>
</div>
<div class="col-xs-9 p-a-1">
    <div *ngFor="let product of products" class="card card-outline-primary">
        <h4 class="card-header">
            {{product.name}}
            <span class="pull-xs-right tag tag-pill tag-primary">
                {{ product.price | currency:"USD":true:"2.2-2" }}
            </span>
        </h4>
        <div class="card-text p-a-1">{{product.description}}</div>
    </div>
</div>
```

В шаблоне появились два новых элемента `button`. Первая — кнопка Home — имеет привязку события, которая вызывает метод `changeCategory` компонента при щелчке на кнопке. Метод не получает аргумента, что равносильно назначению категории `null` и выбору всех товаров.

Привязка `ngFor` применяется к другому элементу `button` с выражением, которое повторяет элемент для каждого значения в массиве, возвращаемого свойством `categories` компонента. Кнопке также назначена привязка события `click`, выражение которой вызывает метод `changeCategory` для выбора текущей категории; это приведет к фильтрации списка товаров, выводимых для пользователя. Также имеется привязка `class`, которая добавляет элемент `button` к активному классу, когда категория, связанная с кнопкой, совпадает с выбранной категорией. Таким образом обеспечивается визуальная обратная связь для пользователя при фильтрации по категориям (рис. 7.6).

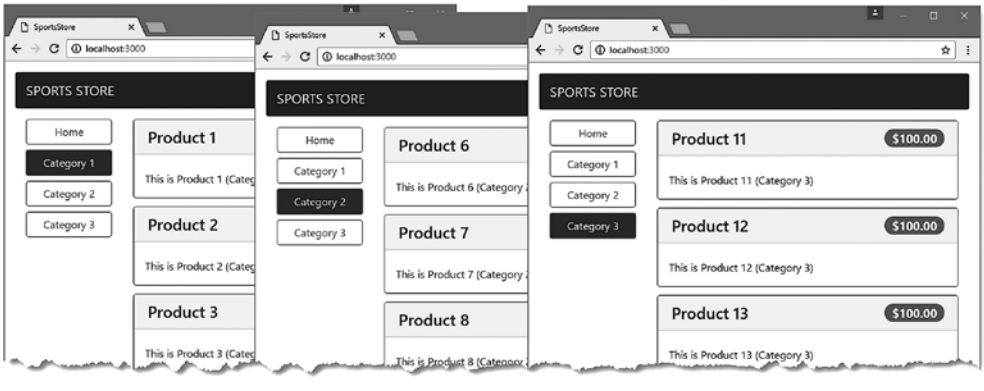


Рис. 7.6. Выбор категории товаров

Страничный вывод списка товаров

Фильтрация продуктов по категориям упрощает работу со списком товаров, но в более типичном решении список разбивается на меньшие фрагменты, и каждый фрагмент выводится на отдельной странице с кнопками навигации для перемещения между страницами.

В листинге 7.20 в компонент магазина вносятся изменения, чтобы в нем сохранялась текущая страница и количество элементов на странице.

Листинг 7.20. Добавление разбивки на страницы в файл store.component.ts

```
import { Component } from "@angular/core";
import { Product } from "../model/product.model";
import { ProductRepository } from "../model/product.repository";

@Component({
    selector: "store",
    moduleId: module.id,
```

```
    templateUrl: "store.component.html"
})
export class StoreComponent {
    public selectedCategory = null;
    public productsPerPage = 4;
    public selectedPage = 1;

    constructor(private repository: ProductRepository) {}

    get products(): Product[] {
        let pageIndex = (this.selectedPage - 1) * this.productsPerPage
        return this.repository.getProducts(this.selectedCategory)
            .slice(pageIndex, pageIndex + this.productsPerPage);
    }

    get categories(): string[] {
        return this.repository.getCategories();
    }

    changeCategory(newCategory?: string) {
        this.selectedCategory = newCategory;
    }

    changePage(newPage: number) {
        this.selectedPage = newPage;
    }

    changePageSize(newSize: number) {
        this.productsPerPage = Number(newSize);
        this.changePage(1);
    }

    get pageNumbers(): number[] {
        return Array(Math.ceil(this.repository
            .getProducts(this.selectedCategory).length / this.productsPerPage))
                .fill(0).map((x, i) => i + 1);
    }
}
```

В этом листинге реализованы две новые возможности: получение страницы с информацией о товарах и изменение размера страниц (с изменением количества товаров, отображаемых на каждой странице).

Здесь есть одна странность, для которой компоненту приходится использовать обходное решение. Встроенная директива `ngFor`, предоставляемая Angular, позволяет генерировать контент только для объектов из массива или коллекции (без использования счетчика). Так как нам нужно сгенерировать пронумерованные кнопки навигации между страницами, приходится создавать массив с нужными числами:

```
...
return Array(Math.ceil(this.repository.getProducts(this.selectedCategory).length
    / this.productsPerPage)).fill(0).map((x, i) => i + 1);
...
```

Эта команда создает новый массив, заполняет его значением 0, а затем при помощи метода `map` генерирует новый массив с числовой последовательностью. Такое решение достаточно хорошо работает в реализации страничного вывода, но выглядит довольно неуклюже; в следующем разделе будет продемонстрировано более удачное решение. В листинге 7.21 приведены изменения в шаблоне компонента магазина, необходимые для реализации страничного вывода.

Листинг 7.21. Реализация страничного вывода в файле store.component.html

```
<div class="navbar navbar-inverse bg-inverse">
    <a class="navbar-brand">SPORTS STORE</a>
</div>
<div class="col-xs-3 p-a-1">
    <button class="btn btn-block btn-outline-primary"
            (click)="changeCategory()">
        Home
    </button>
    <button *ngFor="let cat of categories"
            class="btn btn-outline-primary btn-block"
            [class.active]="cat == selectedCategory"
            (click)="changeCategory(cat)">
        {{cat}}
    </button>
</div>
<div class="col-xs-9 p-a-1">
    <div *ngFor="let product of products" class="card card-outline-primary">
        <h4 class="card-header">
            {{product.name}}
            <span class="pull-xs-right tag tag-pill tag-primary">
                {{ product.price | currency:"USD":true:"2.2-2" }}
            </span>
        </h4>
        <div class="card-text p-a-1">{{product.description}}</div>
    </div>
    <div class="form-inline pull-xs-left m-r-1">
        <select class="form-control" [value]="productsPerPage"
                (change)="changePageSize($event.target.value)">
            <option value="3">3 per Page</option>
            <option value="4">4 per Page</option>
            <option value="6">6 per Page</option>
            <option value="8">8 per Page</option>
        </select>
    </div>

    <div class="btn-group pull-xs-right">
        <button *ngFor="let page of pageNumbers" (click)="changePage(page)"
            class="btn btn-outline-primary" [class.active]="page == selectedPage">
                {{page}}
        </button>
    </div>
</div>
```

В разметку добавляется элемент `select`, позволяющий изменять размер страницы, и набор кнопок для перехода между страницами товаров. Новые элементы содержат привязки данных, связывающие их со свойствами и методами, предоставляемыми компонентом. В результате мы получаем список товаров, с которым удобнее работать (рис. 7.7).

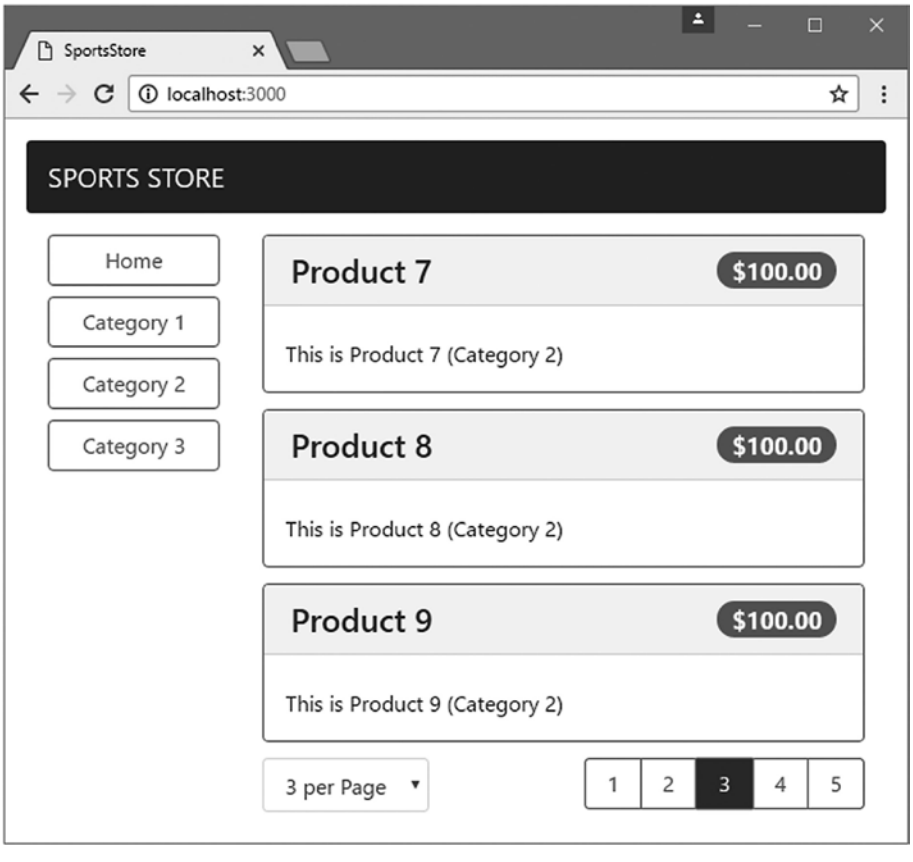


Рис. 7.7. Страничный вывод списка товаров

ПРИМЕЧАНИЕ

Элемент select в листинге 7.21 заполняется элементами option, которые определяются статически (вместо создания на основе данных компонента). Как следствие, при передаче выбранного значения методу changePageSize будет использоваться тип string; этим объясняется необходимость преобразования строки в число перед назначением размера страницы в листинге 7.20. Будьте внимательны при получении данных от элементов HTML; следите за тем, чтобы они относились к ожидаемому типу. Аннотации типов TypeScript в этой ситуации не помогут, потому что выражение привязки данных вычисляется во время выполнения — после того, как компилятор TypeScript сгенерирует код JavaScript, не содержащий дополнительной информации типа.

Создание нестандартной директивы

В этом разделе мы создадим нестандартную директиву, чтобы нам не приходилось генерировать массив, заполненный числами, для создания кнопок навигации. Angular предоставляет хороший набор встроенных директив, но разработчик может относительно просто создавать собственные директивы для решения задач, присущих его приложению, или для поддержки возможностей, отсутствующих во встроенных директивах. Создайте файл counter.directive.ts в папке SportsStore/app/store и используйте его для определения класса из листинга 7.22.

Листинг 7.22. Содержимое файла counter.directive.ts в папке SportsStore/src/app/store

```
import {
    Directive, ViewContainerRef, TemplateRef, Input, Attribute, SimpleChanges
} from "@angular/core";

@Directive({
    selector: "[counterOf]"
})
export class CounterDirective {

    constructor(private container: ViewContainerRef,
        private template: TemplateRef<Object>) {
    }

    @Input("counterOf")
    counter: number;

    ngOnChanges(changes: SimpleChanges) {
        this.container.clear();
        for (let i = 0; i < this.counter; i++) {
            this.container.createEmbeddedView(this.template,
                new CounterDirectiveContext(i + 1));
        }
    }
}

class CounterDirectiveContext {
    constructor(public $implicit: any) { }
}
```

Это пример *структурной директивы*, более подробно рассматриваемой в главе 16. Такие директивы применяются к элементам через свойство `counter` и используют специальные средства Angular для многократного создания контента (по аналогии со встроенной директивой `ngFor`). В этом случае вместо того, чтобы возвращать каждый объект в коллекции, нестандартная директива возвращает серию чисел, которые могут использоваться для создания кнопок навигации между страницами.

ПРИМЕЧАНИЕ

При изменении количества страниц директива удаляет весь созданный контент и начинает все заново. В более сложных директивах этот процесс может быть достаточно затратным; в главе 16 я расскажу, как улучшить быстродействие.

Чтобы использовать директиву, ее необходимо добавить в свойство `declarations` функционального модуля, как показано в листинге 7.23.

Листинг 7.23. Регистрация нестандартной директивы в файле store.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule } from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { StoreComponent } from "./store.component";
import { CounterDirective } from "./counter.directive";

@NgModule({
    imports: [ModelModule, BrowserModule, FormsModule],
    declarations: [StoreComponent, CounterDirective],
    exports: [StoreComponent]
})
export class StoreModule { }
```

После того как директива была зарегистрирована, она может использоваться в шаблоне компонента магазина для замены директивы `ngFor`, как показано в листинге 7.24.

Листинг 7.24. Замена встроенной директивы в файле store.component.html

```
<div class="navbar navbar-inverse bg-inverse">
    <a class="navbar-brand">SPORTS STORE</a>
</div>
<div class="col-xs-3 p-a-1">
    <button class="btn btn-block btn-outline-primary"
            (click)="changeCategory()">
        Home
    </button>
    <button *ngFor="let cat of categories"
            class="btn btn-outline-primary btn-block"
            [class.active]="cat == selectedCategory"
            (click)="changeCategory(cat)">
        {{cat}}
    </button>
</div>
<div class="col-xs-9 p-a-1">
    <div *ngFor="let product of products" class="card card-outline-primary">
        <h4 class="card-header">
            {{product.name}}
            <span class="pull-xs-right tag tag-pill tag-primary">
                {{ product.price | currency:"USD":true:"2.2-2" }}
            </span>
        </h4>
        <div class="card-text p-a-1">{{product.description}}</div>
    </div>
    <div class="form-inline pull-xs-left m-r-1">
        <select class="form-control" [value]="productsPerPage"
                (change)="changePageSize($event.target.value)">
```

```
            <option value="3">3 per Page</option>
            <option value="4">4 per Page</option>
            <option value="6">6 per Page</option>
            <option value="8">8 per Page</option>
        </select>
    </div>

    <div class="btn-group pull-xs-right">
    <button *counter="let page of pageCount" (click)="changePage(page)"
        class="btn btn-outline-primary" [class.active]="page == selectedPage">
                {{page}}
        </button>
    </div>
</div>
```

Новая привязка данных зависит от настройки нестандартной директивы с использованием свойства `pageCount`. В листинге 7.25 массив чисел заменяется простым значением `number`, предоставляющим результат выражения.

Листинг 7.25. Поддержка нестандартной директивы в файле store.component.ts

```
import { Component } from "@angular/core";
import { Product } from "../model/product.model";
import { ProductRepository } from "../model/product.repository";

@Component({
    selector: "store",
    moduleId: module.id,
    templateUrl: "store.component.html"
})
export class StoreComponent {
    public selectedCategory = null;
    public productsPerPage = 4;
    public selectedPage = 1;

    constructor(private repository: ProductRepository) {}

    get products(): Product[] {
        let pageIndex = (this.selectedPage - 1) * this.productsPerPage
        return this.repository.getProducts(this.selectedCategory)
            .slice(pageIndex, pageIndex + this.productsPerPage);
    }

    get categories(): string[] {
        return this.repository.getCategories();
    }

    changeCategory(newCategory?: string) {
        this.selectedCategory = newCategory;
    }

    changePage(newPage: number) {
        this.selectedPage = newPage;
```

```
    }

    changePageSize(newSize: number) {
        this.productsPerPage = Number(newSize);
        this.changePage(1);
    }

    get pageCount(): number {
        return Math.ceil(this.repository
            .getProducts(this.selectedCategory).length / this.productsPerPage)
    }

    //get pageNumbers(): number[] {
    //    return Array(Math.ceil(this.repository
    //        .getProducts(this.selectedCategory).length / this.productsPerPage))
    //            .fill(0).map((x, i) => i + 1);
    //}
}
```

В приложении SportsStore внешне ничего не изменилось, но этот раздел продемонстрировал, что встроенная функциональность Angular может дополняться нестандартным кодом, адаптированным для потребностей конкретного проекта.

Итоги

В этой главе мы начали работу над проектом SportsStore. Начало главы было посвящено «закладке фундамента» для проекта: установке и настройке средств разработчика, созданию корневых структурных блоков для приложения и началу работы над функциональными модулями. Когда подготовка была завершена, мы смогли быстро добавлять новые возможности для вывода фиктивной модели данных, реализации разбивки на страницы и фильтрации товаров по категориям. Глава завершается созданием нестандартной директивы для демонстрации того, как встроенные функции Angular могут расширяться специализированным кодом. В следующей главе работа над приложением SportsStore будет продолжена.

8 SportsStore: выбор товаров и оформление заказа

В этой главе я продолжу добавлять новые возможности в приложение SportsStore, созданное в главе 7. Мы добавим поддержку корзины и процесса оформления заказа и заменим фиктивные данные данными от REST-совместимой веб-службы.

Подготовка приложения

Эта глава не требует подготовки; в ней мы продолжим использование проекта SportsStore из главы 7. Выполните следующую команду из папки SportsStore, чтобы запустить REST-совместимую веб-службу:

```
npm start json
```

Откройте второе окно командной строки и введите следующую команду из папки SportsStore, чтобы запустить инструменты разработки и сервер HTTP:

```
ng serve --port 3000 --open
```

ПРИМЕЧАНИЕ

В бесплатном архиве исходного кода, который можно загрузить на сайте apress.com, содержатся версии проекта SportsStore для каждой главы (если вы не хотите начинать работу с главы 7).

Создание корзины

Пользователю понадобится корзина, в которую помещаются продукты для последующего оформления заказа. В этом разделе мы добавим функциональность корзины в приложение и интегрируем ее в магазин, чтобы пользователь мог выбирать интересующие его товары.

Создание модели корзины

Исходной точкой для функциональности корзины станет новый класс модели, который будет использоваться для сбора товаров, выбранных пользователем.

Создайте файл с именем cart.model.ts в папке SportsStore/src/app/model и включите в него определение класса из листинга 8.1.

Листинг 8.1. Содержимое файла cart.model.ts в папке SportsStore/app/model

```
import { Injectable } from "@angular/core";
import { Product } from "./product.model";

@Injectable()
export class Cart {
    public lines: CartLine[] = [];
    public itemCount: number = 0;
    public cartPrice: number = 0;

    addLine(product: Product, quantity: number = 1) {
        let line = this.lines.find(line => line.product.id == product.id);
        if (line != undefined) {
            line.quantity += quantity;
        } else {
            this.lines.push(new CartLine(product, quantity));
        }
        this.recalculate();
    }

    updateQuantity(product: Product, quantity: number) {
        let line = this.lines.find(line => line.product.id == product.id);
        if (line != undefined) {
            line.quantity = Number(quantity);
        }
        this.recalculate();
    }

    removeLine(id: number) {
        let index = this.lines.findIndex(line => line.product.id == id);
        this.lines.splice(index);
        this.recalculate();
    }

    clear() {
        this.lines = [];
        this.itemCount = 0;
        this.cartPrice = 0;
    }

    private recalculate() {
        this.itemCount = 0;
        this.cartPrice = 0;
        this.lines.forEach(l => {
            this.itemCount += l.quantity;
            this.cartPrice += (l.quantity * l.product.price);
        })

    }
```

```
}

export class CartLine {

    constructor(public product: Product,
        public quantity: number) {}

    get lineTotal() {
        return this.quantity * this.product.price;
    }
}
```

Выбранные продукты представляются массивом объектов `CartLine`, каждый из которых содержит объект `Product` и количество единиц товара. В классе `Cart` хранится общее количество выбранных товаров и их общая стоимость, которая будет отображаться в процессе покупки.

Во всем приложении должен использоваться всего один объект `Cart`, который гарантирует, что любая часть приложения сможет получить информацию о товарах, выбранных пользователем. Для этого мы оформим `Cart` в виде службы; это означает, что Angular будет отвечать за создание экземпляра класса `Cart` и использовать его, когда потребуется создать компонент с аргументом конструктора `Cart`. Это еще один пример использования механизма внедрения зависимостей Angular, который может использоваться для совместного доступа к объектам в приложении, он подробно рассматривается в главах 19 и 20. Декоратор `@Injectable`, который применяется к классу `Cart` в листинге, означает, что класс будет использоваться как служба. (Строго говоря, декоратор `@Injectable` обязателен только при наличии у класса собственных аргументов конструктора; тем не менее его лучше применять всегда, потому что он сигнализирует, что класс предназначен для использования в качестве службы.) Листинг 8.2 регистрирует класс `Cart` в качестве службы в свойстве `providers` класса функционального модуля модели.

Листинг 8.2. Регистрация Cart как службы в файле model.module.ts

```
import { NgModule } from "@angular/core";
import { ProductRepository } from "./product.repository";
import { StaticDataSource } from "./static.datasource";
import { Cart } from "./cart.model";

@NgModule({
    providers: [ProductRepository, StaticDataSource, Cart]
})
export class ModelModule { }
```

Создание компонентов для сводной информации корзины

Компоненты являются основными структурными блоками в приложениях Angular, потому что они позволяют легко создавать изолированные блоки кода и контента. Приложение SportsStore выводит сводную информацию о выбранных товарах в заголовке страницы; для реализации этой функциональности мы

создадим компонент. Создайте файл с именем cartSummary.component.ts в папке SportsStore/src/app/store и определите в нем компонент из листинга 8.3.

Листинг 8.3. Содержимое файла cartSummary.component.ts в папке SportsStore/src/app/store

```
import { Component } from "@angular/core";
import { Cart } from "../model/cart.model";

@Component({
    selector: "cart-summary",
    moduleId: module.id,
    templateUrl: "cartSummary.component.html"
})
export class CartSummaryComponent {

    constructor(public cart: Cart) { }
}
```

Когда потребуется создать экземпляр этого компонента, среда Angular должна предоставить объект `Cart` как аргумент конструктора с использованием службы, настроенной в предыдущем разделе (добавлением класса `Cart` в свойство `providers` функционального модуля). В варианте поведения служб по умолчанию один объект `Cart` будет создан и использован в приложении, хотя доступны разные варианты поведения служб (см. главу 20).

Чтобы предоставить компоненту шаблон, создайте файл HTML с именем cartSummary.component.html в одной папке с файлом класса компонента и добавьте разметку из листинга 8.4.

Листинг 8.4. Содержимое файла cartSummary.component.html в папке SportsStore/src/app/store

```
<div class="pull-xs-right">
    <small>
        Your cart:
        <span *ngIf="cart.itemCount > 0">
            {{ cart.itemCount }} item(s)
            {{ cart.cartPrice | currency:"USD":true:"2.2-2" }}
        </span>
        <span *ngIf="cart.itemCount == 0">
            (empty)
        </span>
    </small>
    <button class="btn btn-sm bg-inverse" [disabled]="cart.itemCount == 0">
        <i class="fa fa-shopping-cart"></i>
    </button>
</div>
```

Шаблон использует объект `Cart`, предоставленный компонентом, для вывода количества товаров в корзине и их общей стоимости. Также предусмотрена кнопка для запуска процесса оформления заказа, который будет добавлен в приложение позднее.

ПРИМЕЧАНИЕ

Элемент button в листинге 8.4 оформляется с использованием классов, определяемых Font Awesome — пакетом из файла package.json в главе 7. Этот пакет, распространяемый с открытым кодом, предоставляет превосходную поддержку значков в веб-приложениях, в том числе и для корзины, необходимой для приложения SportsStore. За дополнительной информацией обращайтесь по адресу http://fontawesome.io.

Листинг 8.5 регистрирует новый компонент в функциональном модуле магазина, чтобы подготовиться к его использованию в следующем разделе.

Листинг 8.5. Регистрация компонента в файле store.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule } from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { StoreComponent } from "./store.component";
import { CounterDirective } from "./counter.directive";
import { CartSummaryComponent } from "./cartsummary.component";

@NgModule({
    imports: [ModelModule, BrowserModule, FormsModule],
    declarations: [StoreComponent, CounterDirective, CartSummaryComponent],
    exports: [StoreComponent]
})
export class StoreModule { }
```

Интеграция корзины в приложение

Компонент `store` играет ключевую роль в интеграции корзины и ее виджета в приложение. В листинге 8.6 обновляется компонент `store`: в него добавляется конструктор с параметром `Cart`, а также определяется метод для добавления товара в корзину.

Листинг 8.6. Добавление поддержки корзины в файл store.component.ts

```
import { Component } from "@angular/core";
import { Product } from "../model/product.model";
import { ProductRepository } from "../model/product.repository";
import { Cart } from "../model/cart.model";

@Component({
    selector: "store",
    moduleId: module.id,
    templateUrl: "store.component.html"
})
export class StoreComponent {
    public selectedCategory = null;
    public productsPerPage = 4;
    public selectedPage = 1;

    constructor(private repository: ProductRepository,
                private cart: Cart) { }

    get products(): Product[] {
```

```
        let pageIndex = (this.selectedPage - 1) * this.productsPerPage
        return this.repository.getProducts(this.selectedCategory)
            .slice(pageIndex, pageIndex + this.productsPerPage);
    }
    get categories(): string[] {
        return this.repository.getCategories();
    }
    changeCategory(newCategory?: string) {
        this.selectedCategory = newCategory;
    }
    changePage(newPage: number) {
        this.selectedPage = newPage;
    }
    changePageSize(newSize: number) {
        this.productsPerPage = Number(newSize);
        this.changePage(1);
    }
    get pageCount(): number {
        return Math.ceil(this.repository
            .getProducts(this.selectedCategory).length / this.productsPerPage)
    }
    addProductToCart(product: Product) {
        this.cart.addLine(product);
    }
}
```

Чтобы завершить интеграцию корзины в компонент магазина, в листинге 8.7 добавляется элемент, который применяет компонент со сводной информацией корзины в шаблон компонента магазина и добавляет в описание каждого товара кнопку с привязкой события для вызова метода `addProductToCart`.

Листинг 8.7. Применение компонента в файле store.component.html

```
<div class="navbar navbar-inverse bg-inverse">
    <a class="navbar-brand">SPORTS STORE</a>
    <cart-summary></cart-summary>
</div>
<div class="col-xs-3 p-a-1">
    <button class="btn btn-block btn-outline-primary"
            (click)="changeCategory()">
        Home
    </button>
    <button *ngFor="let cat of categories"
            class="btn btn-outline-primary btn-block"
            [class.active]="cat == selectedCategory"
            (click)="changeCategory(cat)">
        {{cat}}
    </button>
</div>
<div class="col-xs-9 p-a-1">
    <div *ngFor="let product of products" class="card card-outline-primary">
        <h4 class="card-header">
```

```
            {{product.name}}
            <span class="pull-xs-right tag tag-pill tag-primary">
                {{ product.price | currency:"USD":true:"2.2-2" }}
            </span>
        </h4>
        <div class="card-text p-a-1">
            {{product.description}}
            <button class="btn btn-success btn-sm pull-xs-right"
                    (click)="addProductToCart(product)">
                Add To Cart
            </button>
        </div>
    </div>
    <div class="form-inline pull-xs-left m-r-1">
        <select class="form-control" [value]="productsPerPage"
                (change)="changePageSize($event.target.value)">
            <option value="3">3 per Page</option>
            <option value="4">4 per Page</option>
            <option value="6">6 per Page</option>
            <option value="8">8 per Page</option>
        </select>
    </div>
    <div class="btn-group pull-xs-right">
        <button *counter="let page of pageCount" (click)="changePage(page)"
            class="btn btn-outline-primary" [class.active]="page == selectedPage">
                {{page}}
        </button>
    </div>
</div>
```

В результате для каждого товара создается кнопка для добавления в корзину (рис. 8.1). Полноценная поддержка корзины еще не реализована, но последствия каждого добавления товара отображаются в сводке в верхней части страницы.

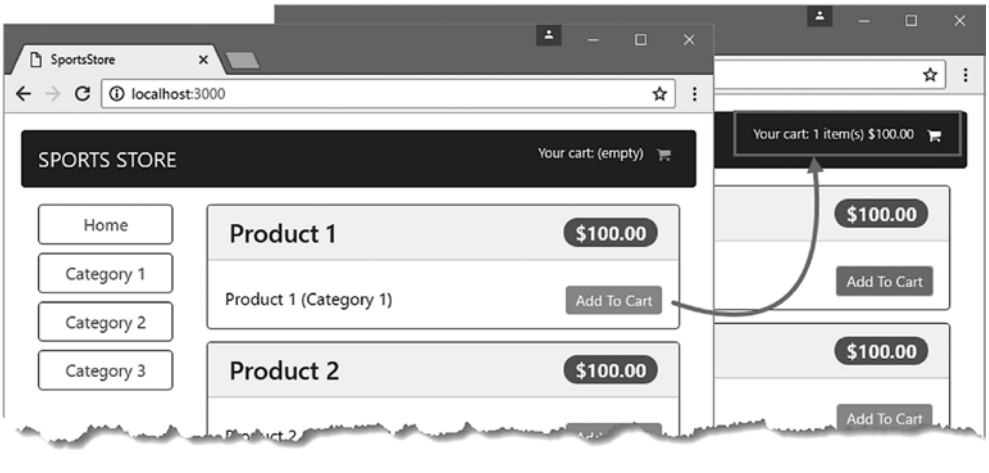


Рис. 8.1. Добавление поддержки корзины в приложение SportsStore

Обратите внимание: при нажатии одной из кнопок Add To Cart содержимое компонента сводки изменяется автоматически. Это стало возможным благодаря тому, что один объект `Cart` совместно используется двумя компонентами и изменения, вносимые одним компонентом, отражаются при вычислении выражений привязок данных в другом компоненте.

Маршрутизация URL

Многие приложения отображают разный контент в разные моменты времени. В приложении SportsStore при нажатии одной из кнопок Add To Cart пользователь должен увидеть подробное описание выбранных товаров, а также получить возможность запустить процесс оформления заказа.

Angular поддерживает механизм *маршрутизации URL*, который использует текущий URL-адрес в браузере, для выбора компонентов, отображаемых для пользователя. Этот механизм упрощает создание приложений, в которых компоненты не имеют жесткого сцепления и легко изменяются без необходимости внесения изменений в других местах. Маршрутизация URL также позволяет легко изменить путь, по которому пользователь взаимодействует с приложением.

В приложении SportsStore мы добавим поддержку трех разных URL-адресов из табл. 8.1. Это очень простая конфигурация, но и такая система маршрутизации обладает широкими возможностями, подробно описанными в главах 25–27.

Таблица 8.1. URL-адреса, поддерживаемые приложением SportsStore

URL	Описание
/store	URL для вывода списка товаров
/cart	URL для вывода корзины
/checkout	URL для процесса оформления заказа

Ниже мы создадим временные компоненты для корзины SportsStore и оформления заказа и интегрируем их в приложение с применением маршрутизации URL. После того как поддержка URL-адресов будет реализована, мы вернемся к компонентам и добавим больше полезных возможностей.

Создание компонентов для содержимого корзины и оформления заказа

Прежде чем добавлять в приложение маршрутизацию URL, необходимо создать компоненты, которые будут отображаться по URL */cart* и */checkout*. Для начала хватит простейшего временного контента — просто чтобы было понятно, какой компонент отображается на странице. Создайте файл с именем cartDetail.component.ts в папке SportsStore/src/app/store и включите в него определение компонента из листинга 8.8.

Листинг 8.8. Содержимое файла cartDetail.component.ts в папке SportsStore/app/store

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

@Component({
    template: `<div><h3 class="bg-info p-a-1">Cart Detail Component</h3></div>`
})
export class CartDetailComponent {}
```

Затем создайте файл checkout.component.ts в папке SportsStore/src/app/store и добавьте определение компонента из листинга 8.9.

Листинг 8.9. Содержимое файла checkout.component.ts в папке SportsStore/app/store

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

@Component({
    template: `<div><h3 class="bg-info p-a-1">Checkout Component</h3></div>`
})
export class CheckoutComponent { }
```

Компонент построен по той же схеме, что и компонент корзины: он выводит временное сообщение, которое наглядно показывает, какой компонент отображается. В листинге 8.10 компоненты регистрируются в функциональном модуле `store` и включаются в свойство `exports`, чтобы они могли использоваться в других местах приложения.

Листинг 8.10. Регистрация компонентов в файле store.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule } from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { StoreComponent } from "./store.component";
import { CounterDirective } from "./counter.directive";
import { CartSummaryComponent } from "./cartsummary.component";
import { CartDetailComponent } from "./cartDetail.component";
import { CheckoutComponent } from "./checkout.component";

@NgModule({
    imports: [ModelModule, BrowserModule, FormsModule],
    declarations: [StoreComponent, CounterDirective, CartSummaryComponent,
        CartDetailComponent, CheckoutComponent],
    exports: [StoreComponent, CartDetailComponent, CheckoutComponent]
})
export class StoreModule { }
```

Создание и применение конфигурации маршрутизации

Теперь, когда у нас имеется набор компонентов, на следующем шаге создается конфигурация маршрутизации, которая описывает соответствия между URL и компонентами. Каждое соответствие между URL и компонентом называется *маршрутом URL*, или просто *маршрутом* (route). Когда мы займемся созданием

более сложных конфигураций маршрутизации, маршруты будут определяться в отдельном файле, но в этом проекте используется другое решение — определение маршрутов в декораторе `@NgModule` корневого модуля приложения (рис. 8.11).

ПРИМЕЧАНИЕ

Механизм маршрутизации Angular требует присутствия в документе HTML элемента base, определяющего базовый URL-адрес, к которому применяются маршруты. Этот элемент был добавлен в главе 7, когда мы создавали проект SportsStore. Если элемент пропущен, Angular сообщит об ошибке и не сможет применить маршруты.

Листинг 8.11. Создание конфигурации маршрутизации в файле app.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { AppComponent } from "./app.component";
import { StoreModule } from "./store/store.module";
import { StoreComponent } from "./store/store.component";
import { CheckoutComponent } from "./store/checkout.component";
import { CartDetailComponent } from "./store/cartDetail.component";
import { RouterModule } from "@angular/router";

@NgModule({
    imports: [BrowserModule, StoreModule,
        RouterModule.forRoot([
            { path: "store", component: StoreComponent },
            { path: "cart", component: CartDetailComponent },
            { path: "checkout", component: CheckoutComponent },
            { path: "**", redirectTo: "/store" }
        ])],
    declarations: [AppComponent],
    bootstrap: [AppComponent]
})
export class AppModule { }
```

Методу `RouterModule.forRoot` передается набор маршрутов, каждый из которых связывает URL с компонентом. Первые три маршрута в листинге соответствуют URL из табл. 8.1. Последний маршрут является универсальным — он перенаправляет любой другой URL на */store*, который отображает `StoreComponent`.

При использовании механизма маршрутизации Angular ищет элемент `router-outlet`, определяющий место для поиска компонента, соответствующего текущему URL. В листинге 8.12 элемент `store` шаблона корневого компонента заменяется элементом `router-outlet`.

Листинг 8.12. Определение цели маршрутизации в файле app.component.ts

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

@Component({
    selector: "app",
    template: "<router-outlet></router-outlet>"
})
export class AppComponent { }
```

Angular применяет конфигурацию маршрутизации, когда вы сохраняете изменения, а браузер перезагружает документ HTML. Контент, отображаемый в окне браузера, не изменился, но в адресной строке браузера видно, что конфигурация маршрутизации была успешно применена (рис. 8.2).

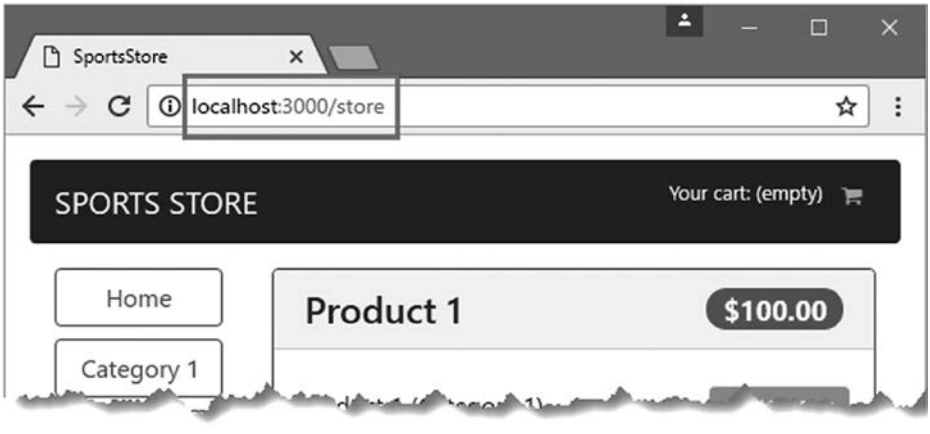


Рис. 8.2. Эффект маршрутизации URL

Навигация в приложении

Когда конфигурация маршрутизации будет настроена, можно переходить к поддержке навигации между компонентами изменением URL-адреса в браузере. Механизм маршрутизации URL зависит от JavaScript API, предоставляемого браузером; это означает, что пользователь не может просто ввести целевой URL-адрес в адресной строке браузера. Вместо этого навигация должна выполняться приложением — либо с использованием кода JavaScript в компоненте (или другом структурном блоке), либо с добавлением атрибутов в элементы HTML в шаблоне.

Когда пользователь щелкает на одной из кнопок Add To Cart, должен отображаться компонент с информацией корзины; это означает, что приложение должно перейти по URL-адресу */cart*. В листинге 8.13 навигация добавляется в метод компонента, который вызывается при нажатии кнопки пользователем.

Листинг 8.13. Навигация с использованием JavaScript в файле store.component.ts

```
import { Component } from "@angular/core";
import { Product } from "../model/product.model";
import { ProductRepository } from "../model/product.repository";
import { Cart } from "../model/cart.model";
import { Router } from "@angular/router";

@Component({
    selector: "store",
    moduleId: module.id,
```

```
    templateUrl: "store.component.html"
})
export class StoreComponent {
    public selectedCategory = null;
    public productsPerPage = 4;
    public selectedPage = 1;

    constructor(private repository: ProductRepository,
        private cart: Cart,
        private router: Router) { }

    get products(): Product[] {
        let pageIndex = (this.selectedPage - 1) * this.productsPerPage
        return this.repository.getProducts(this.selectedCategory)
            .slice(pageIndex, pageIndex + this.productsPerPage);
    }

    get categories(): string[] {
        return this.repository.getCategories();
    }

    changeCategory(newCategory?: string) {
        this.selectedCategory = newCategory;
    }

    changePage(newPage: number) {
        this.selectedPage = newPage;
    }

    changePageSize(newSize: number) {
        this.productsPerPage = Number(newSize);
        this.changePage(1);
    }

    get pageCount(): number {
        return Math.ceil(this.repository
            .getProducts(this.selectedCategory).length / this.productsPerPage)
    }
    addProductToCart(product: Product) {
        this.cart.addLine(product);
        this.router.navigateByUrl("/cart");
    }
}
```

Конструктор получает параметр `Router`, который предоставляется Angular через механизм внедрения зависимостей при создании нового экземпляра компонента. В методе `addProductToCart` метод `Router.navigateByUrl` используется для перехода по URL */cart*.

Навигация также может осуществляться добавлением атрибута `routerLink` в элементы в шаблоне. В листинге 8.14 атрибут `routerLink` применяется к кнопке в шаблоне компонента сводной информации корзины.

Листинг 8.14. Добавление навигационного атрибута в файл cartSummary.component.html

```
<div class="pull-xs-right">
    <small>
        Your cart:
        <span *ngIf="cart.itemCount > 0">
            {{ cart.itemCount }} item(s)
            {{ cart.cartPrice | currency:"USD":true:"2.2-2" }}
        </span>
        <span *ngIf="cart.itemCount == 0">
            (empty)
        </span>
    </small>
    <button class="btn btn-sm bg-inverse" [disabled]="cart.itemCount == 0"
            routerLink="/cart">
        <i class="fa fa-shopping-cart"></i>
    </button>
</div>
```

В качестве значения атрибута `routerLink` указывается URL-адрес, по которому должно переходить приложение при щелчке на кнопке. Эта конкретная кнопка блокируется при пустой корзине, так что переход будет происходить только после добавления товара в корзину пользователем.

Для добавления поддержки атрибута `routerLink` необходимо импортировать модуль `RouterModule` в функциональный модуль, как показано в листинге 8.15.

Листинг 8.15. Импортирование модуля маршрутизации в файл store.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule } from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { StoreComponent } from "./store.component";
import { CounterDirective } from "./counter.directive";
import { CartSummaryComponent } from "./cartsummary.component";
import { CartDetailComponent } from "./cartDetail.component";
import { CheckoutComponent } from "./checkout.component";
import { RouterModule } from "@angular/router";

@NgModule({
    imports: [ModelModule, BrowserModule, FormsModule, RouterModule],
    declarations: [StoreComponent, CounterDirective, CartSummaryComponent,
        CartDetailComponent, CheckoutComponent],
    exports: [StoreComponent, CartDetailComponent, CheckoutComponent]
})
export class StoreModule { }
```

Чтобы увидеть, как работает навигация, сохраните изменения в файлах, а после того как браузер перезагрузит документ HTML, щелкните на одной из кнопок Add To Cart. Браузер переходит по URL-адресу */cart* (рис. 8.3).

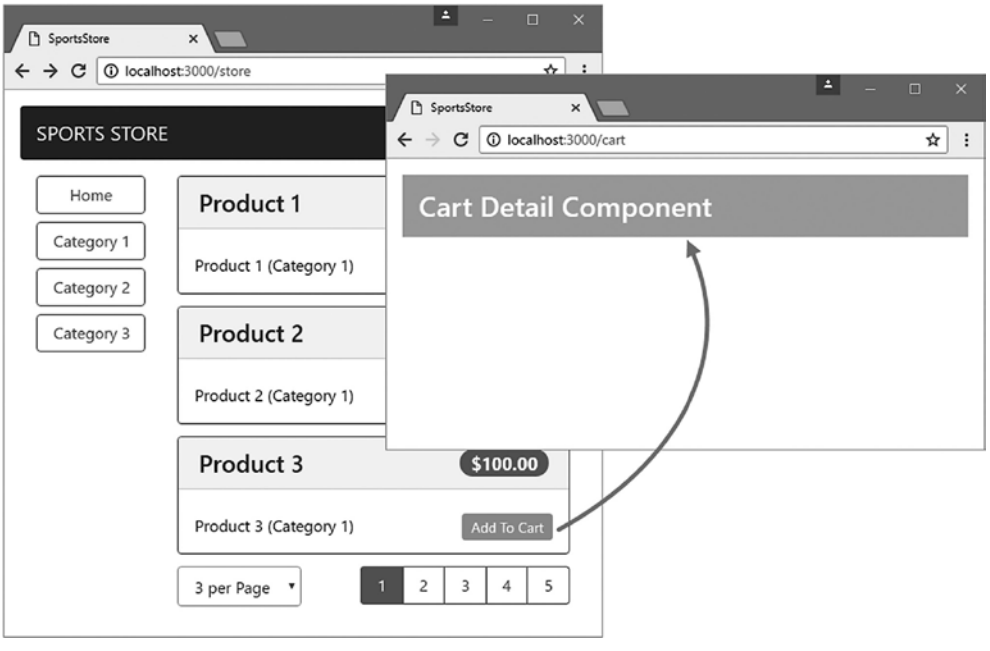


Рис. 8.3. Маршрутизация URL

Защитники маршрутов

Помните, что навигация может выполняться только приложением. Если изменить URL прямо в адресной строке браузера, то браузер запросит введенный URL-адрес у веб-сервера. Пакет `lite-server`, отвечающий на запросы HTTP, отвечает на любой запрос, не соответствующий файлу, возвращая содержимое index.html. Обычно такое поведение удобно, потому что оно предотвращает ошибку HTTP при щелчке на кнопке обновления в браузере.

Но оно также может создать проблемы, если приложение ожидает, что пользователь будет переходить в приложении по определенному пути. Например, если щелкнуть на одной из кнопок Add To Cart, а затем щелкнуть на кнопке обновления в браузере, то сервер HTTP вернет содержимое файла index.html, а Angular немедленно перейдет к компоненту содержимого корзины и пропустит часть приложения, позволяющую пользователю выбирать продукты.

В некоторых приложениях возможность начинать с разных URL-адресов имеет смысл, а для других случаев Angular поддерживает *защитников маршрутов* (route guards), используемых для управления системой маршрутизации.

Чтобы приложение не могло начинать с URL */cart* или */order*, добавьте файл storeFirst.guard.ts в папку SportsStore/app и определите класс из листинга 8.16.

Листинг 8.16. Содержимое файла storeFirst.guard.ts в папке SportsStore/src/app

```
import { Injectable } from "@angular/core";
import {
    ActivatedRouteSnapshot, RouterStateSnapshot,
    Router
} from "@angular/router";
import { StoreComponent } from "./store/store.component";

@Injectable()
export class StoreFirstGuard {
    private firstNavigation = true;

    constructor(private router: Router) { }

    canActivate(route: ActivatedRouteSnapshot,
        state: RouterStateSnapshot): boolean {
        if (this.firstNavigation) {
            this.firstNavigation = false;
            if (route.component != StoreComponent) {
                this.router.navigateByUrl("/");
                return false;
            }
        }
        return true;
    }
}
```

Существуют разные способы защиты маршрутов, описанные в главе 27; эта разновидность защитника предотвращает активизацию маршрута. Она реализуется классом, определяющим метод `canActivate`. Реализация метода использует объекты контекста, предоставляемые Angular; по ним она проверяет, является ли целевым компонентом `StoreComponent`. Если метод `canActivate` вызывается впервые и использоваться должен другой компонент, то метод `Router.navigateByUrl` используется для перехода к корневому URL-адресу.

В листинге применяется декоратор `@Injectable`, потому что защитники маршрутов являются службами. В листинге 8.17 защитник регистрируется в качестве службы при помощи свойства `providers` корневого модуля и защищает каждый маршрут при помощи свойства `canActivate`.

Листинг 8.17. Защита маршрутов в файле app.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { AppComponent } from "./app.component";
import { StoreModule } from "./store/store.module";
import { StoreComponent } from "./store/store.component";
import { CheckoutComponent } from "./store/checkout.component";
import { CartDetailComponent } from "./store/cartDetail.component";
import { RouterModule } from "@angular/router";
import { StoreFirstGuard } from "./storeFirst.guard";

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

```
        RouterModule.forRoot([
            {
                path: "store", component: StoreComponent,
                canActivate: [StoreFirstGuard]
            },
            {
                path: "cart", component: CartDetailComponent,
                canActivate: [StoreFirstGuard]
            },
            {
                path: "checkout", component: CheckoutComponent,
                canActivate: [StoreFirstGuard]
            },
            { path: "**", redirectTo: "/store" }
        ])],
    providers: [StoreFirstGuard],
    declarations: [AppComponent],
    bootstrap: [AppComponent]
})
export class AppModule { }
```

Если обновить браузер после щелчка на одной из кнопок Add To Cart, вы увидите, что браузер автоматически перенаправляется в безопасное состояние (рис. 8.4).

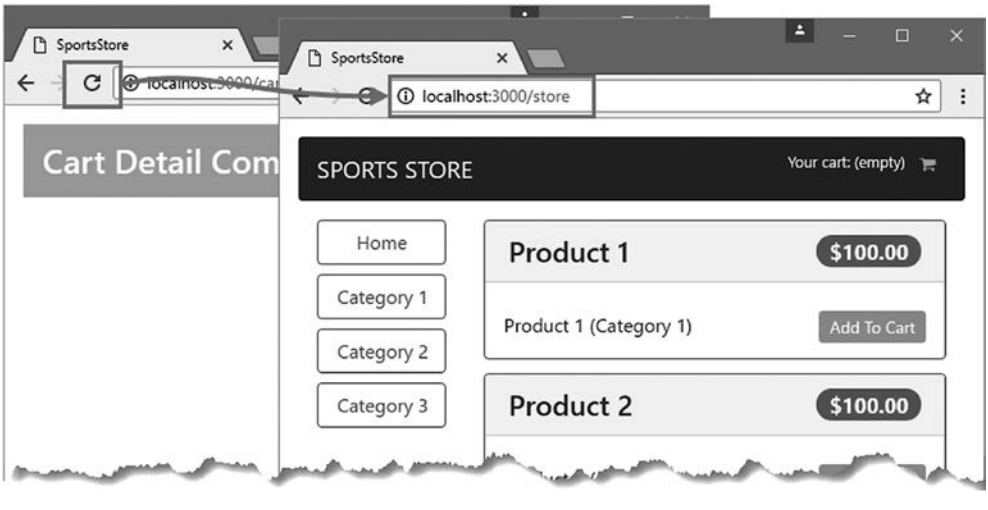


Рис. 8.4. Защита маршрутов

Завершение вывода содержимого корзины

Естественный процесс разработки приложений Angular переходит от подготовки инфраструктуры (например, маршрутизации URL) к реализации функций, видимых пользователю. Теперь, когда в приложении реализована поддержка навигации, приходит время отобразить представление с подробным содержимым

корзины. В листинге 8.18 встроенный шаблон исключается из компонента корзины, вместо него назначается внешний шаблон из того же каталога, а в конструктор добавляется параметр `Cart`, который будет доступен в шаблоне через свойство с именем `cart`.

Листинг 8.18. Изменение шаблона в файле cardDetail.component.ts

```
import { Component } from "@angular/core";
import { Cart } from "../model/cart.model";

@Component({
    moduleId: module.id,
    templateUrl: "cartDetail.component.html"
})
export class CartDetailComponent {
    constructor(public cart: Cart) { }
}
```

Чтобы завершить функциональность корзины, создайте файл HTML с именем cartDetail.component.html в папке SportsStore/src/app/store и добавьте контент из листинга 8.19.

Листинг 8.19. Содержимое файла cardDetail.component.html в папке SportsStore/src/app/store

```
<div class="navbar navbar-inverse bg-inverse">
    <a class="navbar-brand">SPORTS STORE</a>
</div>

<div class="m-a-1">
    <h2 class="text-xs-center">Your Cart</h2>
    <table class="table table-bordered table-striped p-a-1">
        <thead>
            <tr>
                <th>Quantity</th>
                <th>Product</th>
                <th class="text-xs-right">Price</th>
                <th class="text-xs-right">Subtotal</th>
            </tr>
        </thead>
        <tbody>
            <tr *ngIf="cart.lines.length == 0">
                <td colspan="4" class="text-xs-center">
                    Your cart is empty
                </td>
            </tr>
            <tr *ngFor="let line of cart.lines">
                <td>
                    <input type="number" class="form-control-sm"
                           style="width:5em"
                           [value]="line.quantity"
                           (change)="cart.updateQuantity(line.product,
                                $event.target.value)"/>
                </td>
                <td>{{line.product.name}}</td>
```

```
                <td class="text-xs-right">
                    {{line.product.price | currency:"USD":true:"2.2-2"}}
                </td>
                <td class="text-xs-right">
                    {{(line.lineTotal) | currency:"USD":true:"2.2-2" }}
                </td>
                <td class="text-xs-center">
                    <button class="btn btn-sm btn-danger"
                            (click)="cart.removeLine(line.product.id)">
                        Remove
                    </button>
                </td>
            </tr>
        </tbody>
        <tfoot>
            <tr>
                <td colspan="3" class="text-xs-right">Total:</td>
                <td class="text-xs-right">
                    {{cart.cartPrice | currency:"USD":true:"2.2-2"}}
                </td>
            </tr>
        </tfoot>
    </table>
</div>
<div class="text-xs-center">
    <button class="btn btn-primary" routerLink="/store">Continue Shopping</button>
    <button class="btn btn-secondary" routerLink="/checkout"
            [disabled]="cart.lines.length == 0">
        Checkout
    </button>
</div>
```

Шаблон выводит таблицу с товарами, выбранными пользователем. Для каждого товара создается элемент `input`, который может использоваться для изменения количества единиц, и кнопка Remove для удаления товара из корзины. Также создаются две навигационные кнопки для возвращения к списку товаров и перехода к оформлению заказа (рис. 8.5). Сочетание привязок данных Angular и общего объекта `Cart` означает, что любые изменения, вносимые в корзину, приводят к немедленному пересчету цены, а если вы щелкнете на кнопке Continue Shopping, изменения отражаются в компоненте со сводной информацией корзины, отображаемом над списком товаров.

Обработка заказов

Процесс обработки заказов от клиентов — самый важный аспект интернет-магазина. Далее в приложении будет реализована поддержка ввода дополнительной информации пользователем и оформления заказа. Для простоты мы не будем заниматься подробностями взаимодействия с платежными системами, которые обычно представляют собой служебные подсистемы, не относящиеся к приложениям Angular.

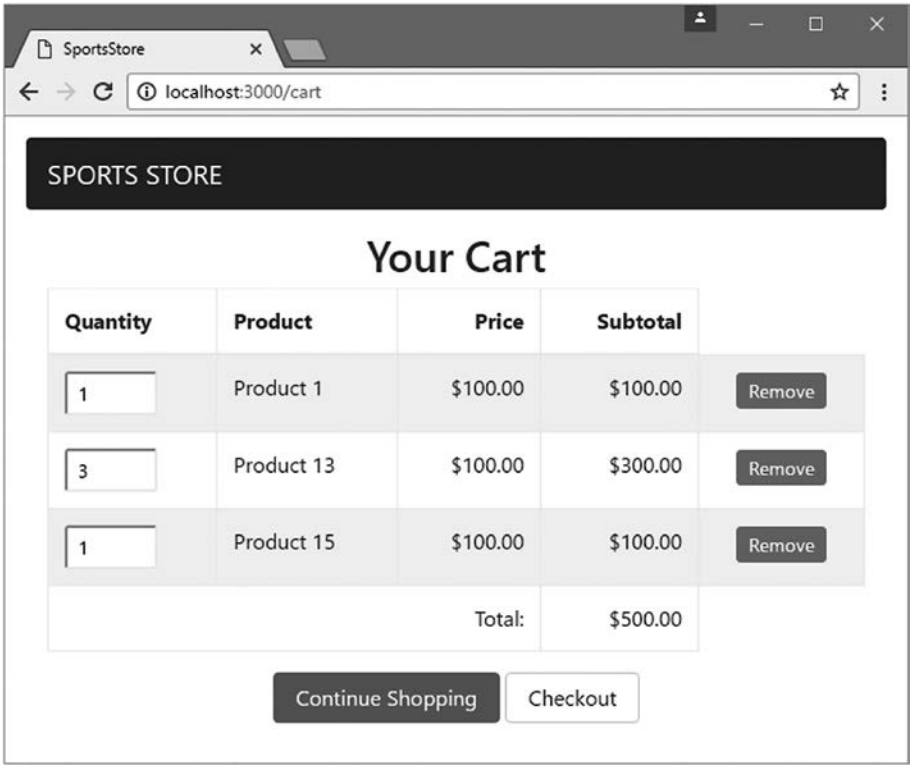


Рис. 8.5. Готовая функциональность корзины

Расширение модели

Для описания заказов, размещенных пользователями, создайте файл с именем order.model.ts в папке SportsStore/src/app/model и добавьте в него код из листинга 8.20.

Листинг 8.20. Содержимое файла order.model.ts в папке SportsStore/src/app/model

```
import { Injectable } from "@angular/core";
import { Cart } from "./cart.model";

@Injectable()
export class Order {
    public id: number;
    public name: string;
    public address: string;
    public city: string;
    public state: string;
    public zip: string;
    public country: string;
    public shipped: boolean = false;
```

```
    constructor(public cart: Cart) { }

    clear() {
        this.id = null;
        this.name = this.address = this.city = null;
        this.state = this.zip = this.country = null;
        this.shipped = false;
        this.cart.clear();
    }
}
```

Класс `Order` также будет оформлен в виде службы; это означает, что он будет существовать всего в одном экземпляре, который будет совместно использоваться в границах приложения. Когда среда Angular создает объект `Order`, она обнаруживает параметр конструктора `Cart` и предоставляет один и тот же объект `Cart`.

Обновление репозитория и источника данных

Для обработки заказов в приложении необходимо расширить репозиторий и источник данных, чтобы они могли получать объекты `Order`. В листинге 8.21 к источнику данных добавляется метод получения заказа. Так как источник данных пока остается фиктивным, метод просто создает на основании заказа строку JSON и выводит ее на консоль JavaScript. Объекты начнут выполнять более полезные функции в следующем разделе, когда мы создадим источник данных, использующий запросы HTTP для взаимодействия с REST-совместимыми веб-службами.

Листинг 8.21. Обработка заказов в файле static.datasource.ts

```
import { Injectable } from "@angular/core";
import { Product} from "./product.model";
import { Observable } from "rxjs/Observable";
import "rxjs/add/observable/from";
import { Order } from "./order.model";

@Injectable()
export class StaticDataSource {
    private products: Product[] = [

       // ...опущено для краткости...

    ];

    getProducts(): Observable<Product[]> {
        return Observable.from([this.products]);
    }

    saveOrder(order: Order): Observable<Order> {
        console.log(JSON.stringify(order));
        return Observable.from([order]);
    }
}
```

Для управления заказами создайте файл order.repository.ts в папке SportsStore/app/model и используйте его для определения класса из листинга 8.22. На данный момент репозиторий содержит всего один метод, но его функциональность будет расширена в главе 9, когда мы займемся созданием средств администрирования.

ПРИМЕЧАНИЕ

Использовать разные репозитории для разных типов модели в приложении не обязательно. Тем не менее я обычно поступаю именно так, потому что один класс, используемый для нескольких типов моделей, становится слишком сложным и неудобным в сопровождении.

Листинг 8.22. Содержимое файла order.repository.ts в папке SportsStore/app/model

```
import { Injectable } from "@angular/core";
import { Observable } from "rxjs/Observable";
import { Order } from "./order.model";
import { StaticDataSource } from "./static.datasource";

@Injectable()
export class OrderRepository {
    private orders: Order[] = [];

    constructor(private dataSource: StaticDataSource) {}

    getOrders(): Order[] {
        return this.orders;
    }

    saveOrder(order: Order): Observable<Order> {
        return this.dataSource.saveOrder(order);
    }
}
```

Обновление функционального модуля

В листинге 8.23 класс `Order` и новый репозиторий регистрируются в качестве служб при помощи свойства `providers` функционального модуля модели.

Листинг 8.23. Регистрация служб в файле model.module.ts

```
import { NgModule } from "@angular/core";
import { ProductRepository } from "./product.repository";
import { StaticDataSource } from "./static.datasource";
import { Cart } from "./cart.model";
import { Order } from "./order.model";
import { OrderRepository } from "./order.repository";

@NgModule({
    providers: [ProductRepository, StaticDataSource, Cart,
                Order, OrderRepository]
})
export class ModelModule { }
```

Получение информации о заказе

Следующим шагом должно стать получение от пользователя дополнительной информации, необходимой для завершения заказа. Angular включает встроенные директивы для работы с формами HTML и проверки их содержимого. В листинге 8.24 происходит подготовка компонента оформления заказа, переключение на внешний шаблон, получение объекта `Order` в параметре конструктора и реализация дополнительной поддержки работы шаблона.

Листинг 8.24. Подготовка формы в файле checkout.component.ts

```
import { Component } from "@angular/core";
import { NgForm } from "@angular/forms";
import { OrderRepository } from "../model/order.repository";
import { Order } from "../model/order.model";

@Component({
    moduleId: module.id,
    templateUrl: "checkout.component.html",
    styleUrls: ["checkout.component.css"]
})
export class CheckoutComponent {
    orderSent: boolean = false;
    submitted: boolean = false;

    constructor(public repository: OrderRepository,
                public order: Order) {}

    submitOrder(form: NgForm) {
        this.submitted = true;
        if (form.valid) {
            this.repository.saveOrder(this.order).subscribe(order => {
                this.order.clear();
                this.orderSent = true;
                this.submitted = false;
            });
        }
    }
}
```

Метод `submitOrder` будет вызываться при отправке данных формой, которая представляется объектом `NgForm`. Если данные, содержащиеся в форме, проходят проверку, то объект `Order` будет передан методу `saveOrder` репозитория, а данные в корзине и заказе сбрасываются.

Свойство `styleUrls` декоратора `@Component` используется для задания одной или нескольких стилевых таблиц CSS, которые должны применяться к контенту шаблона компонента. Чтобы предоставить обратную связь проверки данных для значений, введенных пользователем в элементах формы HTML, создайте файл с именем checkout.component.css в папке SportsStore/app/store и определите стили в листинге 8.25.

Листинг 8.25. Содержимое файла checkout.component.css в папке SportsStore/app/store

```
input.ng-dirty.ng-invalid { border: 2px solid #ff0000 }
input.ng-dirty.ng-valid { border: 2px solid #6bc502 }
```

Angular добавляет элементы к классам `ng-dirty`, `ng-valid` и `ng-valid` для обозначения статуса проверки. Полный набор классов проверки данных описан в главе 14, а пока скажем, что стили в листинге 8.25 добавляют зеленую рамку к элементам `input`, содержащим проверенные данные, и красную рамку — к недействительным элементам.

Последний фрагмент мозаики — шаблон компонента, который предоставляет пользователю поля формы, необходимые для заполнения свойств объекта `Order` (листинг 8.26).

Листинг 8.26. Содержимое файла checkout.component.html в папке SportsStore/src/app/store

```
<div class="navbar navbar-inverse bg-inverse">
    <a class="navbar-brand">SPORTS STORE</a>
</div>
<div *ngIf="orderSent" class="m-a-1 text-xs-center">
    <h2>Thanks!</h2>
    <p>Thanks for placing your order.</p>
    <p>We'll ship your goods as soon as possible.</p>
    <button class="btn btn-primary" routerLink="/store">Return to Store</button>
</div>
<form *ngIf="!orderSent" #form="ngForm" novalidate
      (ngSubmit)="submitOrder(form)" class="m-a-1">
    <div class="form-group">
        <label>Name</label>
        <input class="form-control" #name="ngModel" name="name"
               [(ngModel)]="order.name" required />
        <span  *ngIf="submitted && name.invalid" class="text-danger">
            Please enter your name
        </span>
    </div>
    <div class="form-group">
        <label>Address</label>
        <input class="form-control" #address="ngModel" name="address"
               [(ngModel)]="order.address" required />
        <span  *ngIf="submitted && address.invalid" class="text-danger">
            Please enter your address
        </span>
    </div>
    <div class="form-group">
        <label>City</label>
        <input class="form-control" #city="ngModel" name="city"
               [(ngModel)]="order.city" required />
        <span  *ngIf="submitted && city.invalid" class="text-danger">
            Please enter your city
        </span>
    </div>
```

```
    <div class="form-group">
        <label>State</label>
        <input class="form-control" #state="ngModel" name="state"
               [(ngModel)]="order.state" required />
        <span  *ngIf="submitted && state.invalid" class="text-danger">
            Please enter your state
        </span>
    </div>
    <div class="form-group">
        <label>Zip/Postal Code</label>
        <input class="form-control" #zip="ngModel" name="zip"
               [(ngModel)]="order.zip" required />
        <span  *ngIf="submitted && zip.invalid" class="text-danger">
            Please enter your zip/postal code
        </span>
    </div>
    <div class="form-group">
        <label>Country</label>
        <input class="form-control" #country="ngModel" name="country"
               [(ngModel)]="order.country" required />
        <span  *ngIf="submitted && country.invalid" class="text-danger">
            Please enter your country
        </span>
    </div>
    <div class="text-xs-center">
        <button class="btn btn-secondary" routerLink="/cart">Back</button>
        <button class="btn btn-primary" type="submit">Complete Order</button>
    </div>
</form>
```

Элементы `form` и `input` в этом шаблоне используют средства Angular, чтобы убедиться в том, что пользователь ввел значение для каждого поля, и предоставляют визуальную обратную связь, если пользователь щелкнул на кнопке Complete Order без заполнения формы. Одна часть этой обратной связи обеспечивается применением стилей, определенных в листинге 8.25, а другая — элементами `span`, которые остаются скрытыми, пока пользователь не попытается отправить недействительную форму.

ПРИМЕЧАНИЕ

Проверка наличия обязательных значений — всего лишь один из способов проверки полей формы в Angular. Как будет показано в главе 14, вы также можете легко реализовать собственную, нестандартную проверку данных.

Чтобы увидеть, как работает этот процесс, начните со списка товаров и щелкните на одной из кнопок Add To Cart, чтобы добавить товар в корзину. Щелкните на кнопке Checkout; на экране появляется форма HTML, изображенная на рис. 8.6. Попробуйте щелкнуть на кнопке Complete Order, не вводя текст ни в одном поле, и вы получите сообщение об ошибке проверки данных. Заполните форму и щелкните на кнопке Complete Order; появляется подтверждающее сообщение (рис. 8.6).

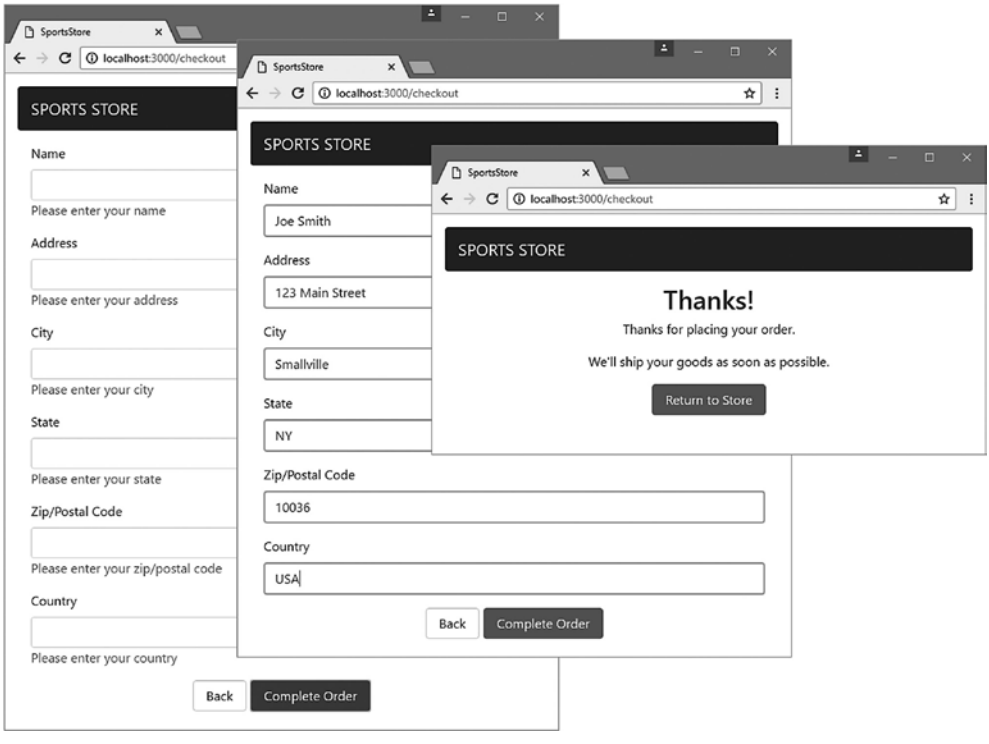


Рис. 8.6. Завершение заказа

На консоли JavaScript в браузере выводится представление заказа в формате JSON:

```
{"cart":
    {"lines":[
        {"product":{"id":1,"name":"Product 1","category":"Category 1",
         "description":"Product 1 (Category 1)","price":100},"quantity":1}],
         "itemCount":1,"cartPrice":100},
    "shipped":false,
    "name":"Joe Smith","address":"123 Main Street",
    "city":"Smallville","state":"NY","zip":"10036","country":"USA"
}
```

Использование REST-совместимой веб-службы

Теперь, когда базовая функциональность SportsStore подготовлена, приходит время заменить фиктивный источник данных другим, получающим данные из REST-совместимой веб-службы, созданной во время подготовки проекта в главе 9.

Чтобы создать источник данных, создайте файл rest.datasource.ts в папке SportsStore/src/app/model и добавьте код в листинге 8.27.

Листинг 8.27. Содержимое файла rest.datasource.ts в папке SportsStore/app/model

```
import { Injectable } from "@angular/core";
import { Http, Request, RequestMethod } from "@angular/http";
import { Observable } from "rxjs/Observable";
import { Product } from "./product.model";
import { Cart } from "./cart.model";
import { Order } from "./order.model";
import "rxjs/add/operator/map";

const PROTOCOL = "http";
const PORT = 3500;

@Injectable()
export class RestDataSource {
    baseUrl: string;

    constructor(private http: Http) {
        this.baseUrl = `${PROTOCOL}://${location.hostname}:${PORT}/`;
    }

    getProducts(): Observable<Product[]> {
        return this.sendRequest(RequestMethod.Get, "products");
    }

    saveOrder(order: Order): Observable<Order> {
        return this.sendRequest(RequestMethod.Post, "orders", order);
    }

    private sendRequest(verb: RequestMethod,
            url: string, body?: Product | Order): Observable<Product | Order> {
        return this.http.request(new Request({
            method: verb,
            url: this.baseUrl + url,
            body: body
        })).map(response => response.json());
    }
}
```

Angular предоставляет встроенную службу `Http`, которая используется для создания запросов HTTP. Конструктор `RestDataSource` получает службу `Http` и использует глобальный объект `location`, предоставленный браузером, для определения URL-адреса для отправки запросов; он соответствует порту 3500 на том хосте, с которого было загружено приложение.

Методы, определяемые классом `RestDataSource`, соответствуют методам, определяемым статическим источником данных, и реализуются вызовом метода `sendRequest`, использующего службу `Http` (см. главу 24).

ПРИМЕЧАНИЕ

При получении данных через HTTP может случиться так, что сетевой затор или повышенная нагрузка на сервер задержат обработку запроса и пользователь будет смотреть на приложение, не получившее данных. В главе 27 я покажу, как настроить систему маршрутизации для предотвращения таких проблем.

Применение источника данных

В завершение этой главы мы применим REST-совместимый источник данных. Для этого мы перестроим приложение, чтобы переход с фиктивных данных на REST-данные осуществлялся с изменениями в одном файле. В листинге 8.28 поведение источника данных изменяется в функциональном модуле модели.

Листинг 8.28. Изменение конфигурации службы в файле model.module.ts

```
import { NgModule } from "@angular/core";
import { ProductRepository } from "./product.repository";
import { StaticDataSource } from "./static.datasource";
import { Cart } from "./cart.model";
import { Order } from "./order.model";
import { OrderRepository } from "./order.repository";
import { RestDataSource } from "./rest.datasource";
import { HttpModule } from "@angular/http";

@NgModule({
    imports: [HttpModule],
    providers: [ProductRepository, Cart, Order, OrderRepository,
        { provide: StaticDataSource, useClass: RestDataSource }]
})
export class ModelModule { }
```

Свойство `imports` используется для объявления зависимости от функционального модуля `HttpModule`, который предоставляет службу `Http`, используемую в листинге 8.27. Изменение в свойстве `providers` сообщает Angular, что когда потребуется создать экземпляр класса с параметром конструктора `StaticDataSource`, вместо него следует использовать `RestDataSource`. Так как оба объекта определяют одинаковые методы, благодаря динамической системе типов JavaScript замена проходит гладко. После того как все изменения будут сохранены, а браузер перезагрузит приложение, фиктивные данные заменяются данными, полученными через HTTP (рис. 8.7).

Если пройти процедуру выбора товаров и оформления заказа, вы сможете убедиться, что источник данных записал заказ в веб-службу; перейдите по следующему URL-адресу:

http://localhost:3500/db

На экране появляется полное содержимое базы данных, включая коллекцию заказов. Вы не сможете выдать запрос с URL */orders*, потому что он требует аутентификации, настройкой которой мы займемся в следующей главе.

ПРИМЕЧАНИЕ

Не забудьте, что данные, предоставленные REST-совместимой веб-службой, сбрасываются при перезапуске npm, с возвратом к данным, определенным в главе 7.

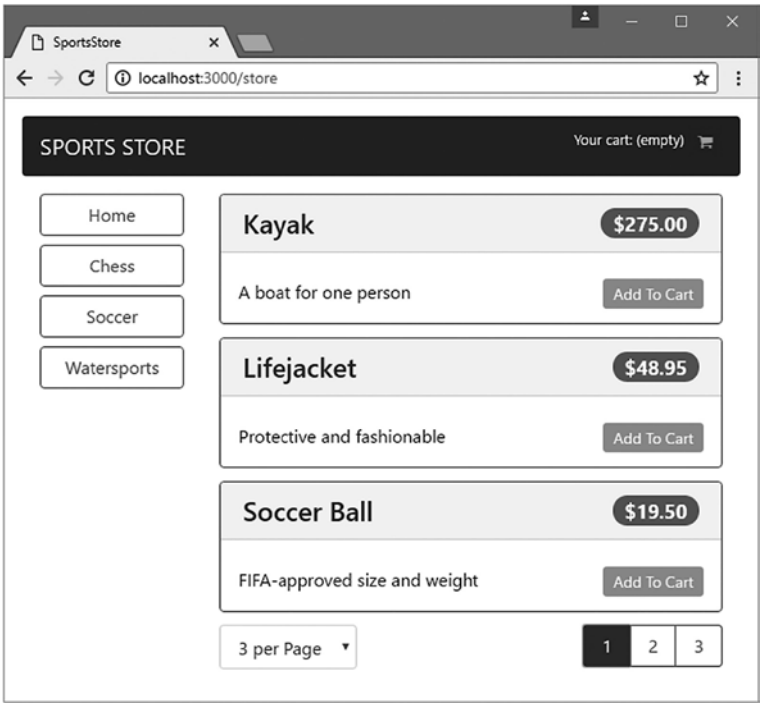


Рис. 8.7. Использование REST-совместимой веб-службы

Итоги

В этой главе мы продолжили расширять функциональность приложения SportsStore: в него была добавлена поддержка корзины для отбора товаров пользователем и процесса оформления заказа, завершающего процесс покупки. В завершающей части главы фиктивный источник данных был заменен источником, отправляющим запросы HTTP REST-совместимой веб-службе. В следующей главе мы займемся созданием механизма администрирования для управления данными SportsStore.

9 SportsStore: администрирование

В этой главе работа над построением приложения SportsStore продолжится: мы добавим в него функции администрирования. Доступ к функциям администрирования понадобится относительно небольшому количеству пользователей, и было бы неэффективно заставлять всех пользователей загружать весь административный код и контент, которые им, скорее всего, не потребуются. Вместо этого функции администрирования будут включены в функциональный модуль, который будет загружаться только в случае надобности. В этом разделе мы подготовим приложение: создадим функциональный модуль, добавим некоторый исходный контент и настроим код приложения так, чтобы модуль загружался динамически.

Подготовка приложения

Эта глава не требует подготовки; в ней мы продолжим использование проекта SportsStore из главы 8. Чтобы запустить REST-совместимую веб-службу, откройте окно командной строки и выполните следующую команду из папки SportsStore:

```
npm run json
```

Откройте второе окно командной строки и введите следующую команду из папки SportsStore, чтобы запустить инструменты разработки и сервер HTTP:

```
ng serve --port 3000 --open
```

ПРИМЕЧАНИЕ

В бесплатном архиве исходного кода, который можно загрузить на сайте apress.com, содержатся версии проекта SportsStore для каждой главы (если вы не хотите начинать работу с предыдущих глав).

Создание модуля

Процесс создания функционального модуля проходит по той же схеме, что и в оставшейся части приложения, не считая одного важного момента: никакая другая часть приложения не зависит от модуля или содержащихся в нем классов, так как это нарушило бы саму концепцию динамической загрузки модуля и заставило модуль JavaScript загружать код администрирования, даже если он не используется.

Работа над функциями администрирования должна начаться с аутентификации. Создайте файл auth.component.ts в папке SportsStore/src/app/admin и определите в нем компонент из листинга 9.1.

Листинг 9.1. Содержимое файла auth.component.ts в папке SportsStore/src/app/admin

```
import { Component } from "@angular/core";
import { NgForm } from "@angular/forms";
import { Router } from "@angular/router";

@Component({
    moduleId: module.id,
    templateUrl: "auth.component.html"
})
export class AuthComponent {
    public username: string;
    public password: string;
    public errorMessage: string;

    constructor(private router: Router) {}

    authenticate(form: NgForm) {
        if (form.valid) {
            // Выполнить аутентификацию
            this.router.navigateByUrl("/admin/main");
        } else {
            this.errorMessage = "Form Data Invalid";
        }
    }
}
```

Компонент определяет свойства для имени пользователя и пароля, которые будут использоваться для аутентификации; свойство `errorMessage`, которое будет выводить сообщения для пользователя при возникновении проблем; и метод `authenticate` для выполнения процесса аутентификации (пока этот метод ничего не делает).

Чтобы создать шаблон для компонента, создайте файл auth.component.html в папке SportsStore/src/app/admin и добавьте в него контент из листинга 9.2.

Листинг 9.2. Содержимое файла auth.component.html в папке SportsStore/src/app/admin

```
<div class="bg-info p-a-1 text-xs-center">
    <h3>SportsStore Admin</h3>
</div>
<div class="bg-danger m-t-1 p-a-1 text-xs-center"
        *ngIf="errorMessage != null">
    {{errorMessage}}
</div>
<div class="p-a-1">
    <form novalidate #form="ngForm" (ngSubmit)="authenticate(form)">
        <div class="form-group">
```

```
            <label>Name</label>
            <input class="form-control" name="username"
                [(ngModel)]="username" required />
        </div>
        <div class="form-group">
            <label>Password</label>
            <input class="form-control" type="password" name="password"
                [(ngModel)]="password" required />
        </div>
        <div class="text-xs-center">
            <button class="btn btn-secondary" routerLink="/">Go back</button>
            <button class="btn btn-primary" type="submit">Log In</button>
        </div>
    </form>
</div>
```

Шаблон содержит форму HTML, которая использует двусторонние выражения привязки данных для свойств компонента. Форма содержит кнопку отправки данных, кнопку для возврата к корневому URL и элемент `div`, видимый только при выводе сообщения об ошибке.

Чтобы создать временную реализацию функций администрирования, создайте файл admin.component.ts в папке SportsStore/src/app/admin и включите в него определение компонента из листинга 9.3.

Листинг 9.3. Содержимое файла admin.component.ts в папке SportsStore/app/admin

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

@Component({
    moduleId: module.id,
    templateUrl: "admin.component.html"
})
export class AdminComponent {}
```

В настоящее время компонент не содержит никакой функциональности. Чтобы определить шаблон для компонента, создайте файл admin.component.html в папке SportsStore/src/app/admin и добавьте временный контент из листинга 9.4.

Листинг 9.4. Содержимое файла admin.component.html в папке SportsStore/src/app/admin

```
<div class="bg-info p-a-1">
    <h3>Placeholder for Admin Features</h3>
</div>
```

Чтобы определить функциональный модуль, создайте файл admin.module.ts в папке SportsStore/app/admin и добавьте код из листинга 9.5.

Листинг 9.5. Содержимое файла admin.module.ts в папке SportsStore/src/app/admin

```
import { NgModule } from "@angular/core";
import { CommonModule } from "@angular/common";
import { FormsModule } from "@angular/forms";
```

```
import { RouterModule } from "@angular/router";
import { AuthComponent } from "./auth.component";
import { AdminComponent } from "./admin.component";

let routing = RouterModule.forChild([
    { path: "auth", component: AuthComponent },
    { path: "main", component: AdminComponent },
    { path: "**", redirectTo: "auth" }
]);

@NgModule({
    imports: [CommonModule, FormsModule, routing],
    declarations: [AuthComponent, AdminComponent]
})
export class AdminModule { }
```

Главное отличие при создании динамически загружаемого модуля заключается в том, что функциональный модуль должен быть автономным и в него должна быть включена вся информация, необходимая для Angular, включая поддерживаемые URL маршрутизации и отображаемые компоненты.

Метод `RouterModule.forChild` используется для определения конфигурации маршрутизации функционального модуля, которая затем включается в свойство `imports` модуля.

Запрет на экспортирование классов из динамически загружаемого модуля не распространяется на импортирование. Этот модуль зависит от функционального модуля модели, который был добавлен в свойство `imports` модуля, чтобы компоненты могли обращаться к репозиториям и классам модели.

Настройка системы маршрутизации URL

Для управления динамически загружаемыми модулями используется конфигурация маршрутизации, которая инициирует процесс загрузки при переходе приложения к конкретному URL-адресу. Листинг 9.6 расширяет конфигурацию маршрутизации приложения, чтобы по URL */admin* загружался функциональный модуль администрирования.

Листинг 9.6. Настройка динамически загружаемого модуля в файле app.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { AppComponent } from "./app.component";
import { StoreModule } from "./store/store.module";
import { StoreComponent } from "./store/store.component";
import { CheckoutComponent } from "./store/checkout.component";
import { CartDetailComponent } from "./store/cartDetail.component";
import { RouterModule } from "@angular/router";
import { StoreFirstGuard } from "./storeFirst.guard";

@NgModule({
```

```
    imports: [BrowserModule, StoreModule,
        RouterModule.forRoot([
            {
                path: "store", component: StoreComponent,
                canActivate: [StoreFirstGuard]
            },
            {
                path: "cart", component: CartDetailComponent,
                canActivate: [StoreFirstGuard]
            },
            {
                path: "checkout", component: CheckoutComponent,
                canActivate: [StoreFirstGuard]
            },
            {
                path: "admin",
                loadChildren: "app/admin/admin.module#AdminModule",
                canActivate: [StoreFirstGuard]
            },
            { path: "**", redirectTo: "/store" }
        ])],
    providers: [StoreFirstGuard],
    declarations: [AppComponent],
    bootstrap: [AppComponent]
})
export class AppModule { }
```

Новый маршрут сообщает Angular, что при переходе приложения на URL */admin* следует загрузить функциональный модуль, определяемый классом с именем `AdminModule` из файла /app/admin/admin.module.ts. Во время обработки административного модуля Angular встраивает содержащуюся в нем маршрутную информацию в общий набор маршрутов.

Переход по URL администрирования

Осталось сделать последний подготовительный шаг — предоставить пользователю возможность перейти на URL */admin*, чтобы загрузился функциональный модуль администрирования, а компонент появился на экране. В листинге 9.7 в шаблон компонента магазина добавляется кнопка для перехода.

Листинг 9.7. Добавление кнопки навигации в файл store.component.html

```
<div class="navbar navbar-inverse bg-inverse">
    <a class="navbar-brand">SPORTS STORE</a>
    <cart-summary></cart-summary>
</div>
<div class="col-xs-3 p-a-1">
    <button class="btn btn-block btn-outline-primary"
            (click)="changeCategory()">
        Home
    </button>
```

```
    <button *ngFor="let cat of categories"
            class="btn btn-outline-primary btn-block"
            [class.active]="cat == selectedCategory"
            (click)="changeCategory(cat)">
        {{cat}}
    </button>
    <button class="btn btn-block btn-danger m-t-3" routerLink="/admin">
        Admin
    </button>
</div>
<div class="col-xs-9 p-a-1">

    <!-- ...элементы опущены для краткости... -->

</div>
```

Чтобы увидеть результат, используйте средства разработчика F12 в браузере для просмотра сетевых запросов, выданных браузером при загрузке приложения. Файлы модуля администрирования не будут загружаться, пока пользователь не щелкнет на кнопке Admin; в этот момент Angular запрашивает файлы и выводит страницу ввода учетных данных (рис. 9.1).

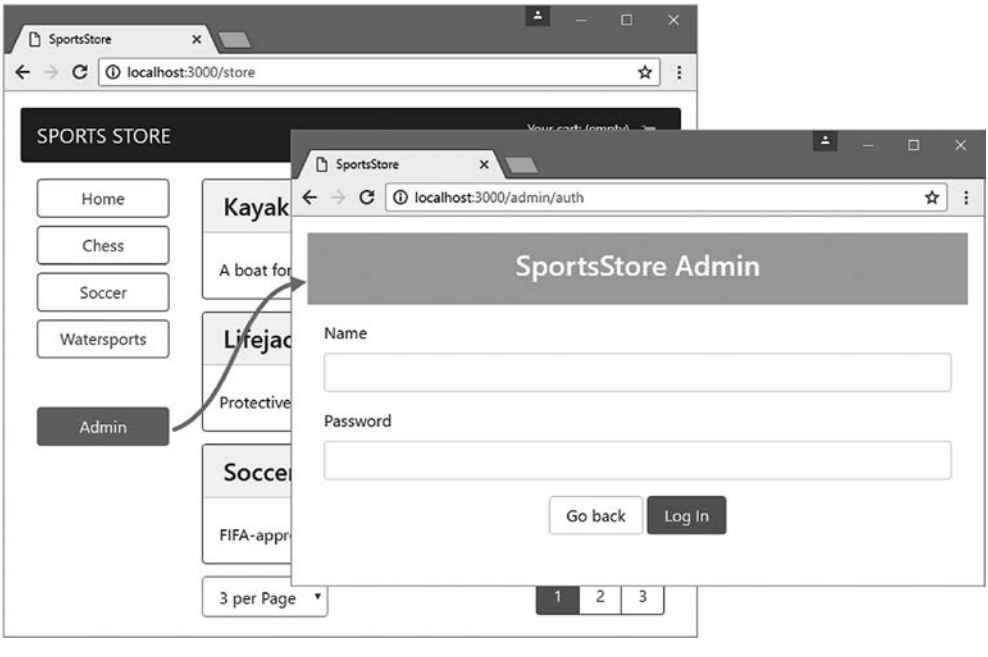


Рис. 9.1. Использование динамически загружаемого модуля

Введите имя и пароль в полях формы. Щелкните на кнопке Log In, чтобы просмотреть временный контент (рис. 9.2). Если хотя бы одно из полей останется пустым, выводится предупреждающее сообщение.

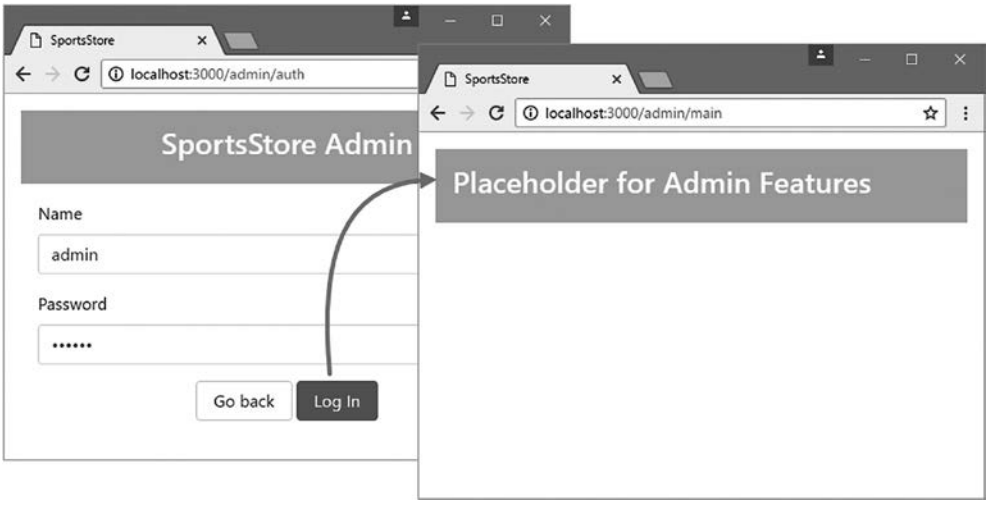


Рис. 9.2. Временная замена функций администрирования

Реализация аутентификации

REST-совместимая веб-служба была настроена так, чтобы она требовала аутентификации для запросов, необходимых средствам администрирования. Ниже модель данных будет расширена для поддержки аутентифицированных запросов HTTP и интегрирована в модуль администрирования.

Система аутентификации

Результатом аутентификации с REST-совместимой веб-службой является веб-маркер JSON (JWT, JSON Web Token), который возвращается сервером и должен включаться во все последующие запросы, тем самым показывая, что приложению разрешается выполнение защищенных операций. Со спецификацией JWT можно ознакомиться по адресу *https://tools.ietf.org/html/rfc7519*, а пока в контексте приложения SportsStore достаточно знать, что приложение Angular может провести аутентификацию отправкой запроса POST на URL */login*, включая в тело запроса объект в формате JSON со свойствами `name` и `password`. В коде аутентификации из главы 7 существует только один набор действительных учетных данных, приведенный в табл. 9.1.

Таблица 9.1. Учетные данные аутентификации, поддерживаемые REST-совместимой веб-службой

Имя пользователя	Пароль
admin	secret

Как упоминалось в главе 7, в реальных проектах учетные данные не должны жестко кодироваться в реальных проектах, но для приложения SportsStore понадобятся именно эти имя и пароль.

Если на URL `/login` отправлены правильные учетные данные, то ответ от REST-совместимой веб-службы будет содержать объект JSON следующего вида:

```
{
  "success": true,
  "token":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJkYXRhIjoiYWRtaW4iLCJleHBpcmVz
           SW4iOiIxaCIsImlhdCI6MTQ3ODk1NjI1Mn0.lJaDDrSu-bHBtdWrz0312p_DG5tKypGv6cA
           NgOyzlg8"
}
```

Свойство `success` описывает результат операции аутентификации, а свойство `token` содержит маркер JWT, который должен включаться в последующие запросы при помощи заголовка HTTP `Authorization` в следующем формате:

```
Authorization: Bearer<eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJkYXRhIjoiYWRtaW4iLC
                      JleHBpcmVzSW4iOiIxaCIsImlhdCI6MTQ3ODk1NjI1Mn0.lJaDDrSu-bHBtd
                      Wrz0312p_DG5tKypGv6cANgOyzlg8>
```

Я настроил маркеры JWT, возвращаемые сервером, чтобы срок их действия истекал через 1 час.

Если серверу были отправлены неверные учетные данные, то объект JSON, возвращенный в ответе, будет содержать только свойство `success` со значением `false`:

```
{
  "success": false
}
```

Расширение источника данных

Большая часть работы будет выполняться классом REST-совместимой веб-службы, потому что этот класс отвечает за отправку запросов аутентификации на URL */login* и включение JWT в последующие запросы. В листинге 9.8 в класс `RestDataSource` добавляется аутентификация, а метод `sendRequest` расширяется для включения JWT в запросы.

Листинг 9.8. Включение аутентификации в файл rest.datasource.ts

```
import { Injectable } from "@angular/core";
import { Http, Request, RequestMethod } from "@angular/http";
import { Observable } from "rxjs/Observable";
import { Product } from "./product.model";
import { Cart } from "./cart.model";
import { Order } from "./order.model";
import "rxjs/add/operator/map";

const PROTOCOL = "http";
const PORT = 3500;
```

```
@Injectable()
export class RestDataSource {
    baseUrl: string;
    auth_token: string;

    constructor(private http: Http) {
        this.baseUrl = `${PROTOCOL}://${location.hostname}:${PORT}/`;
    }

    authenticate(user: string, pass: string): Observable<boolean> {
        return this.http.request(new Request({
            method: RequestMethod.Post,
            url: this.baseUrl + "login",
            body: { name: user, password: pass }
        })).map(response => {
            let r = response.json();
            this.auth_token = r.success ? r.token : null;
            return r.success;
        });
    }

    getProducts(): Observable<Product[]> {
        return this.sendRequest(RequestMethod.Get, "products");
    }

    saveOrder(order: Order): Observable<Order> {
        return this.sendRequest(RequestMethod.Post,
            "orders", order);
    }

    private sendRequest(verb: RequestMethod,
            url: string, body?: Product | Order, auth: boolean = false)
                : Observable<Product | Product[] | Order | Order[]> {

        let request = new Request({
            method: verb,
            url: this.baseUrl + url,
            body: body
        });
        if (auth && this.auth_token != null) {
            request.headers.set("Authorization", `Bearer<${this.auth_token}>`);
        }
        return this.http.request(request).map(response => response.json());
    }
}
```

Создание службы аутентификации

Вместо того чтобы открывать доступ к источнику данных всему коду приложения, мы создадим службу, которая будет использоваться для выполнения аутентификации и проверки ее прохождения. Создайте файл auth.service.ts в папке SportsStore/src/app/model и определите класс из листинга 9.9.

Листинг 9.9. Содержимое файла auth.service.ts в папке SportsStore/src/app/model

```
import { Injectable } from "@angular/core";
import { Observable } from "rxjs/Observable";
import { RestDataSource } from "./rest.datasource";
import "rxjs/add/operator/map";

@Injectable()
export class AuthService {

    constructor(private datasource: RestDataSource) {}

    authenticate(username: string, password: string): Observable<boolean> {
        return this.datasource.authenticate(username, password);
    }

    get authenticated(): boolean {
        return this.datasource.auth_token != null;
    }

    clear() {
        this.datasource.auth_token = null;
    }
}
```

Метод `authenticate` получает учетные данные пользователя и передает их методу `authenticate` источника данных, возвращая объект `Observable`, который возвращает `true`, если процесс аутентификации завершился успешно, или `false` в противном случае. Свойство `authenticated`, доступное только для чтения, возвращает `true`, если источник данных получил маркер аутентификации. Метод `clear` удаляет маркер из источника данных.

Листинг 9.10 регистрирует новую службу в функциональном модуле модели. Он также добавляет запись `providers` для класса `RestDataSource`, который в предшествующих главах использовался только как замена для класса `StaticDataSource`. Так как класс `AuthService` имеет параметр конструктора `RestDataSource`, ему потребуется собственная запись в модуле.

Листинг 9.10. Конфигурация служб в файле model.module.ts

```
import { NgModule } from "@angular/core";
import { ProductRepository } from "./product.repository";
import { StaticDataSource } from "./static.datasource";
import { Cart } from "./cart.model";
import { Order } from "./order.model";
import { OrderRepository } from "./order.repository";
import { RestDataSource } from "./rest.datasource";
import { HttpModule } from "@angular/http";
import { AuthService } from "./auth.service";

@NgModule({
    imports: [HttpModule],
    providers: [ProductRepository, Cart, Order, OrderRepository,
        { provide: StaticDataSource, useClass: RestDataSource },
```

```
        RestDataSource, AuthService]
})
export class ModelModule { }
```

Включение аутентификации

На следующем шаге компонент, получающий учетные данные от пользователя, подключается для выполнения аутентификации через новую службу, как показано в листинге 9.11.

Листинг 9.11. Включение аутентификации в файле auth.component.ts

```
import { Component } from "@angular/core";
import { NgForm } from "@angular/forms";
import { Router } from "@angular/router";
import { AuthService } from "../model/auth.service";

@Component({
    moduleId: module.id,
    templateUrl: "auth.component.html"
})
export class AuthComponent {
    public username: string;
    public password: string;
    public errorMessage: string;

    constructor(private router: Router,
                private auth: AuthService) { }

    authenticate(form: NgForm) {
        if (form.valid) {
            this.auth.authenticate(this.username, this.password)
                .subscribe(response => {
                    if (response) {
                        this.router.navigateByUrl("/admin/main");
                    }
                    this.errorMessage = "Authentication Failed";
                })
        } else {
            this.errorMessage = "Form Data Invalid";
        }
    }
}
```

Чтобы приложение не допускало прямых переходов к функциям администрирования, что привело бы к отправке запросов HTTP без маркера, создайте файл auth.guard.ts в папке SportsStore/src/app/admin и определите защитника маршрута из листинга 9.12.

Листинг 9.12. Содержимое файла auth.guard.ts в папке SportsStore/app/admin

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

```
import { AuthService } from "../model/auth.service";

@Injectable()
export class AuthGuard {

    constructor(private router: Router,
                private auth: AuthService) { }

    canActivate(route: ActivatedRouteSnapshot,
        state: RouterStateSnapshot): boolean {

        if (!this.auth.authenticated) {
            this.router.navigateByUrl("/admin/auth");
            return false;
        }
        return true;
    }
}
```

В листинге 9.13 защитник маршрута применяется к одному из маршрутов, определяемых функциональным модулем администрирования.

Листинг 9.13. Защита маршрута в файле admin.module.ts

```
import { NgModule } from "@angular/core";
import { CommonModule } from "@angular/common";
import { FormsModule } from "@angular/forms";
import { RouterModule } from "@angular/router";
import { AuthComponent } from "./auth.component";
import { AdminComponent } from "./admin.component";
import { AuthGuard } from "./auth.guard";

let routing = RouterModule.forChild([
    { path: "auth", component: AuthComponent },
    { path: "main", component: AdminComponent, canActivate: [AuthGuard] },
    { path: "**", redirectTo: "auth" }
]);

@NgModule({
    imports: [CommonModule, FormsModule, routing],
    providers: [AuthGuard],
    declarations: [AuthComponent, AdminComponent]
})
export class AdminModule {}
```

Чтобы протестировать систему аутентификации, щелкните на кнопке Admin, введите учетные данные и щелкните на кнопке Log In. Если ввести учетные данные из табл. 9.1, то вы увидите временный контент функций администрирования. Если ввести другие учетные данные, появится сообщение об ошибке. Оба результата представлены на рис. 9.3.

ПРИМЕЧАНИЕ

Долгосрочное хранение маркера не осуществляется. Перезагрузите приложение в браузере, чтобы начать все заново и опробовать другие учетные данные.

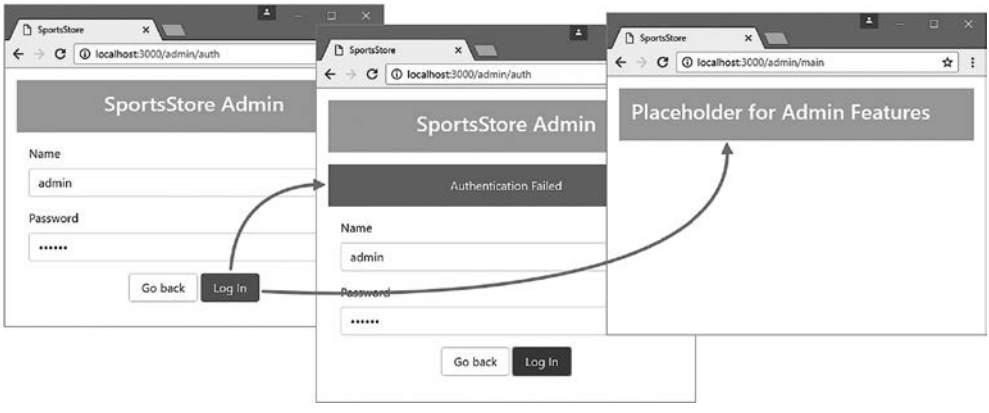


Рис. 9.3. Тестирование функций администрирования

Расширение источника данных и репозиториев

После того как система аутентификации займет свое место, следующим шагом должно стать расширение источника данных для отправки аутентифицированных запросов и предоставления доступа к этим функциям через классы заказов и репозиториев товаров.

Листинг 9.14 добавляет к источнику данных методы, включающие маркер аутентификации.

Листинг 9.14. Добавление новых операций в файле rest.datasource.ts

```
import { Injectable } from "@angular/core";
import { Http, Request, RequestMethod } from "@angular/http";
import { Observable } from "rxjs/Observable";
import { Product } from "./product.model";
import { Cart } from "./cart.model";
import { Order } from "./order.model";
import "rxjs/add/operator/map";

const PROTOCOL = "http";
const PORT = 3500;

@Injectable()
export class RestDataSource {
    baseUrl: string;
    auth_token: string;

    constructor(private http: Http) {
        this.baseUrl = `${PROTOCOL}://${location.hostname}:${PORT}/`;
    }

    authenticate(user: string, pass: string): Observable<boolean> {
```

```
    return this.http.request(new Request({
        method: RequestMethod.Post,
        url: this.baseUrl + "login",
        body: { name: user, password: pass }
    })).map(response => {
        let r = response.json();
        this.auth_token = r.success ? r.token : null;
        return r.success;
    });
}

getProducts(): Observable<Product[]> {
    return this.sendRequest(RequestMethod.Get, "products");
}

saveProduct(product: Product): Observable<Product> {
    return this.sendRequest(RequestMethod.Post, "products",
        product, true);
}

updateProduct(product): Observable<Product> {
    return this.sendRequest(RequestMethod.Put,
        `products/${product.id}`, product, true);
}

deleteProduct(id: number): Observable<Product> {
    return this.sendRequest(RequestMethod.Delete,
        `products/${id}`, null, true);
}

getOrders(): Observable<Order[]> {
    return this.sendRequest(RequestMethod.Get,
        "orders", null, true);
}

deleteOrder(id: number): Observable<Order> {
    return this.sendRequest(RequestMethod.Delete,
        `orders/${id}`, null, true);
}

updateOrder(order: Order): Observable<Order> {
    return this.sendRequest(RequestMethod.Put,
        `orders/${order.id}`, order, true);
}

saveOrder(order: Order): Observable<Order> {
    return this.sendRequest(RequestMethod.Post,
        "orders", order);
}

private sendRequest(verb: RequestMethod,
        url: string, body?: Product | Order, auth: boolean = false)
            : Observable<Product | Product[] | Order | Order[]> {

    let request = new Request({
        method: verb,
        url: this.baseUrl + url,
```

```
            body: body
        });
        if (auth && this.auth_token != null) {
            request.headers.set("Authorization", `Bearer<${this.auth_token}>`);
        }
        return this.http.request(request).map(response => response.json());
    }
}
```

В листинге 9.15 в класс репозитория товаров добавляются новые методы для создания, обновления или удаления товаров. Метод `saveProduct` отвечает за создание и обновление товаров; этот подход хорошо работает при использовании одного объекта, находящегося под управлением компонента (вы убедитесь в этом позднее). В листинге аргументу конструктора также назначается новый тип `RestDataSource`.

Листинг 9.15. Добавление новых операций в файл product.repository.ts

```
import { Injectable } from "@angular/core";
import { Product } from "./product.model";
import { RestDataSource } from "./rest.datasource";

@Injectable()
export class ProductRepository {
    private products: Product[] = [];
    private categories: string[] = [];

    constructor(private dataSource: RestDataSource) {
        dataSource.getProducts().subscribe(data => {
            this.products = data;
            this.categories = data.map(p => p.category)
                .filter((c, index, array) => array.indexOf(c) == index).sort();
        });
    }

    getProducts(category: string = null): Product[] {
        return this.products
            .filter(p => category == null || category == p.category);
    }

    getProduct(id: number): Product {
        return this.products.find(p => p.id == id);
    }

    getCategories(): string[] {
        return this.categories;
    }

    saveProduct(product: Product) {
        if (product.id == null || product.id == 0) {
            this.dataSource.saveProduct(product)
                .subscribe(p => this.products.push(p));
        } else {
            this.dataSource.updateProduct(product)
```

```
                .subscribe(p => {
                    this.products.splice(this.products.
                        findIndex(p => p.id == product.id), 1, product);
                });
        }
    }

    deleteProduct(id: number) {
        this.dataSource.deleteProduct(id).subscribe(p => {
            this.products.splice(this.products.
                findIndex(p => p.id == id), 1);
        })
    }
}
```

В листинге 9.16 соответствующие изменения вносятся в репозиторий заказов, с добавлением метода для модификации и удаления заказов.

Листинг 9.16. Добавление новых операций в файл order.repository.ts

```
import { Injectable } from "@angular/core";
import { Observable } from "rxjs/Observable";
import { Order } from "./order.model";
import { RestDataSource } from "./rest.datasource";

@Injectable()
export class OrderRepository {
    private orders: Order[] = [];
    private loaded: boolean = false;

    constructor(private dataSource: RestDataSource) {}

    loadOrders() {
        this.loaded = true;
        this.dataSource.getOrders()
            .subscribe(orders => this.orders = orders);
    }

    getOrders(): Order[] {
        if (!this.loaded) {
            this.loadOrders();
        }
        return this.orders;
    }

    saveOrder(order: Order): Observable<Order> {
        return this.dataSource.saveOrder(order);
    }

    updateOrder(order: Order) {
        this.dataSource.updateOrder(order).subscribe(order => {
            this.orders.splice(this.orders.
                findIndex(o => o.id == order.id), 1, order);
        });
    }
```

```
    deleteOrder(id: number) {
        this.dataSource.deleteOrder(id).subscribe(order => {
            this.orders.splice(this.orders.findIndex(o => id == o.id));
        });
    }
}
```

Репозиторий заказов определяет метод `loadOrders`, который получает заказы из репозитория, и гарантирует, что запрос не будет отправлен REST-совместимой веб-службе до выполнения аутентификации.

Создание структуры подсистемы администрирования

Итак, система аутентификации успешно работает, а репозитории предоставляют весь спектр операций. Теперь можно переходить к созданию структуры для отображения функций администрирования, которая будет создана на базе существующей конфигурации маршрутизации URL. В табл. 9.2 перечислены URL-адреса, которые будут поддерживаться, и функциональность, которую каждый из них будет предоставлять пользователю.

Таблица 9.2. URL-адреса функций администрирования

URL	Описание
/admin/main/products	Вывод всех товаров в таблице с кнопками для редактирования и удаления существующих, а также создания новых товаров
/admin/main/products/create	Пустой редактор для создания нового товара
/admin/main/products/edit/1	Редактор для изменения существующего товара (с предварительным заполнением полей)
/admin/main/orders	Список всех заказов в таблице с кнопками для пометки заказа как отправленного и отмены заказа с его удалением

Создание временных компонентов

По моему опыту, самый простой способ расширения функциональности проекта Angular — определение компонентов-заместителей с временным контентом и построение структуры приложения на их основе. После того как структура будет сформирована, я возвращаюсь к компонентам и реализую их функциональность более подробно. Чтобы создать временный компонент для функций администрирования, создайте файл с именем productTable.component.ts в папке SportsStore/src/app/admin и определите компонент в соответствии с листингом 9.17. Этот компонент будет отвечать за вывод списка товаров вместе с кнопками редактирования или удаления отдельных позиций, а также создания нового товара.

Листинг 9.17. Содержимое файла productTable.component.ts в папке SportsStore/app/admin

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

@Component({
    template: `<div class="bg-info p-a-1">
                 <h3>Product Table Placeholder</h3>
               </div>`
})
export class ProductTableComponent {}
```

Создайте файл productEditor.component.ts в папке SportsStore/src/app/admin и используйте его для определения компонента из листинга 9.18. Этот компонент будет обеспечивать ввод пользователем информации, необходимой для создания или редактирования компонента.

Листинг 9.18. Содержимое файла productEditor.component.ts в папке SportsStore/src/app/admin

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

@Component({
    template: `<div class="bg-warning p-a-1">
                 <h3>Product Editor Placeholder</h3>
               </div>`
})
export class ProductEditorComponent { }
```

Чтобы создать компонент для управления заказами, создайте файл orderTable.component.ts в папке SportsStore/src/app/admin и добавьте код из листинга 9.19.

Листинг 9.19. Содержимое файла orderTable.component.ts в папке SportsStore/src/app/admin

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

@Component({
    template: `<div class="bg-primary p-a-1">
                 <h3>Order Table Placeholder</h3>
               </div>`
})
export class OrderTableComponent { }
```

Подготовка общего контента и функционального модуля

Компоненты, созданные в предыдущем разделе, отвечают за конкретные функции. Чтобы свести эти функции воедино и дать пользователю возможность переходить между ними, необходимо изменить шаблон временного компонента, который использовался для демонстрации результата успешной попытки аутентификации. Замените временный контент элементами, представленными в листинге 9.20.

Листинг 9.20. Замена контента в файле admin.component.html

```
<div class="navbar navbar-inverse bg-info">
    <a class="navbar-brand">SPORTS STORE Admin</a>
</div>
<div class="m-t-1">
    <div class="col-xs-3">
        <button class="btn btn-outline-info btn-block"
                routerLink="/admin/main/products"
                routerLinkActive="active">
            Products
        </button>
        <button class="btn btn-outline-info btn-block"
                routerLink="/admin/main/orders"
                routerLinkActive="active">
            Orders
        </button>
        <button class="btn btn-outline-danger btn-block" (click)="logout()">
            Logout
        </button>
    </div>
    <div class="col-xs-9">
        <router-outlet></router-outlet>
    </div>
</div>
```

Шаблон содержит элемент `router-outlet`, который будет использоваться для отображения компонентов из предыдущего раздела. Также имеются кнопки для перехода по URL /admin/main/products и /admin/main/orders, на которых выбираются товары или осуществляется управление заказами. Эти кнопки используют атрибут `routerLinkActive`, который добавляет элемент к классу CSS при активности маршрута, заданного атрибутом `routerLink`.

Шаблон также содержит кнопку Logout с привязкой события для метода с именем `logout`. Листинг 9.21 добавляет в компонент этот метод, использующий службу аутентификации для удаления маркера носителя и перевода приложения на URL по умолчанию.

Листинг 9.21. Реализация метода logout в файле admin.component.ts

```
import { Component } from "@angular/core";
import { Router } from "@angular/router";
import { AuthService } from "../model/auth.service";

@Component({
    moduleId: module.id,
    templateUrl: "admin.component.html"
})
export class AdminComponent {

    constructor(private auth: AuthService,
                private router: Router) { }

    logout() {
```

```
        this.auth.clear();
        this.router.navigateByUrl("/");
    }
}
```

Листинг 9.22 подключает временные компоненты, которые будут использоваться для всех функций администрирования, и расширяет конфигурацию маршрутизации URL для URL из табл. 9.2.

Листинг 9.22. Настройка функционального модуля в файле admin.module.ts

```
import { NgModule } from "@angular/core";
import { CommonModule } from "@angular/common";
import { FormsModule } from "@angular/forms";
import { RouterModule } from "@angular/router";
import { AuthComponent } from "./auth.component";
import { AdminComponent } from "./admin.component";
import { AuthGuard } from "./auth.guard";
import { ProductTableComponent } from "./productTable.component";
import { ProductEditorComponent } from "./productEditor.component";
import { OrderTableComponent } from "./orderTable.component";

let routing = RouterModule.forChild([
    { path: "auth", component: AuthComponent },
    {
        path: "main", component: AdminComponent, canActivate: [AuthGuard],
        children: [
            { path: "products/:mode/:id", component: ProductEditorComponent },
            { path: "products/:mode", component: ProductEditorComponent },
            { path: "products", component: ProductTableComponent },
            { path: "orders", component: OrderTableComponent },
            { path: "**", redirectTo: "products" }
        ]
    },
    { path: "**", redirectTo: "auth" }
]);

@NgModule({
    imports: [CommonModule, FormsModule, routing],
    providers: [AuthGuard],
    declarations: [AuthComponent, AdminComponent,
        ProductTableComponent, ProductEditorComponent, OrderTableComponent]
})
export class AdminModule {}
```

Отдельные маршруты могут расширяться при помощи свойства `children`, которое используется для определения маршрутов, направленных на вложенный элемент `router-outlet`; эта возможность рассматривается в главе 25. Как вы вскоре увидите, компоненты могут получать информацию об активном маршруте от Angular, что позволяет им адаптировать свое поведение. Маршруты могут включать параметры маршрутов (такие, как `:mode` или `:id`), которые используются для передачи информации компонентам с целью изменения их поведения.

Когда все изменения будут сохранены, щелкните на кнопке Admin и введите имя `admin` с паролем `secret`. Вы увидите новый макет, показанный на рис. 9.4. Кнопки Products и Orders изменяют компонент, отображаемый элементом `router-outlet` из листинга 9.20, а кнопка Logout осуществляет выход из административного подраздела.

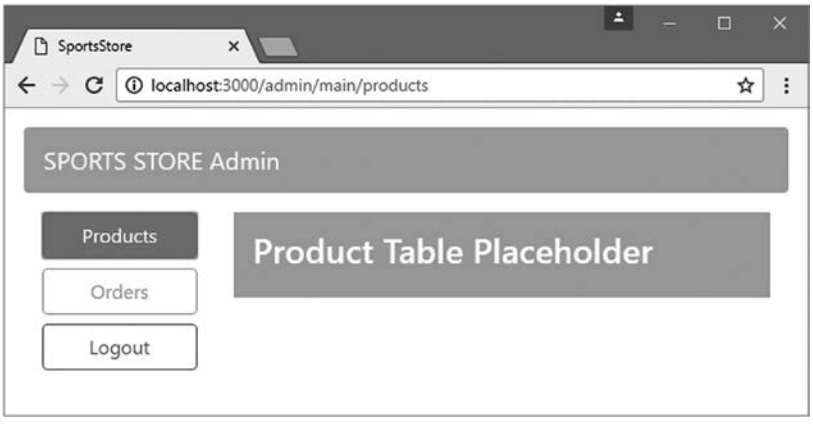


Рис. 9.4. Структура административного макета

Реализация работы с товарами

Первое, что видит пользователь в административном интерфейсе, — список товаров с возможностью создания нового товара, удаления или редактирования уже существующих товаров. В листинге 9.23 временный контент удаляется из компонента таблицы товаров и в него добавляется логика реализации этой возможности.

Листинг 9.23. Замена временного контента в файле productTable.component.ts

```
import { Component } from "@angular/core";
import { Product } from "../model/product.model";
import { ProductRepository } from "../model/product.repository";

@Component({
    moduleId: module.id,
    templateUrl: "productTable.component.html"
})
export class ProductTableComponent {

    constructor(private repository: ProductRepository) { }

    getProducts(): Product[] {
        return this.repository.getProducts();
    }

    deleteProduct(id: number) {
        this.repository.deleteProduct(id);
    }
}
```

Методы компонента предоставляют доступ к товарам в репозитории, равно как и возможность удаления товаров. Другие операции реализуются компонентом редактирования, который активизируется с использованием URL маршрутизации в шаблоне компонента. Создайте файл productTable.component.html в папке SportsStore/src/app/admin и включите в него разметку из листинга 9.24.

Листинг 9.24. Файл productTable.component.html в папке SportsStore/app/admin

```
<table class="table table-sm table-striped">
    <thead>
        <tr>
            <th>ID</th><th>Name</th><th>Category</th><th>Price</th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        <tr *ngFor="let p of getProducts()">
            <td>{{p.id}}</td>
            <td>{{p.name}}</td>
            <td>{{p.category}}</td>
            <td>{{p.price | currency:"USD":true:"2.2-2"}}</td>
            <td>
                <button class="btn btn-sm btn-warning"
                        [routerLink]="['/admin/main/products/edit', p.id]">
                    Edit
                </button>
                <button class="btn btn-sm btn-danger" (click)="deleteProduct(p.id)">
                    Delete
                </button>
            </td>
        </tr>
    </tbody>
</table>
<button class="btn btn-primary" routerLink="/admin/main/products/create">
    Create New Product
</button>
```

Шаблон содержит таблицу, которая использует директиву `ngFor` для генерирования строки для каждого товара, возвращаемого методом `getProducts` компонента. Каждая строка содержит кнопку Delete, которая вызывает метод `delete` компонента, и кнопку Edit для перехода к URL-адресу, связанному с компонентом редактора. Компонент редактора также является целью кнопки Create New Product, хотя в этом случае используется другой URL-адрес.

Реализация редактора товара

Компоненты могут получать информацию о текущем URL маршрутизации и соответствующим образом адаптировать свое поведение. Компонент редактора использует эту возможность для того, чтобы различать запросы на создание нового и редактирование существующего товара. В листинге 9.25 компонент редактора дополняется функциональностью создания или редактирования товара.

Листинг 9.25. Добавление функциональности в файл productEditor.component.ts

```
import { Component } from "@angular/core";
import { Router, ActivatedRoute } from "@angular/router";
import { NgForm } from "@angular/forms";
import { Product } from "../model/product.model";
import { ProductRepository } from "../model/product.repository";

@Component({
    moduleId: module.id,
    templateUrl: "productEditor.component.html"
})
export class ProductEditorComponent {
    editing: boolean = false;
    product: Product = new Product();

    constructor(private repository: ProductRepository,
                private router: Router,
                activeRoute: ActivatedRoute) {

        this.editing = activeRoute.snapshot.params["mode"] == "edit";
        if (this.editing) {
            Object.assign(this.product,
                repository.getProduct(activeRoute.snapshot.params["id"]));
        }
    }

    save(form: NgForm) {
        this.repository.saveProduct(this.product);
        this.router.navigateByUrl("/admin/main/products");
    }
}
```

Angular передает объект `ActivatedRoute` в аргументе конструктора при создании нового экземпляра класса компонента; он может использоваться для анализа активизированного маршрута. В этом случае компонент определяет, должен он создать или изменить товар, и в случает редактирования читает текущую информацию из репозитория. Также имеется метод `save`, который использует репозиторий для сохранения изменений, внесенных пользователем.

Чтобы определить шаблон для компонента, создайте файл productEditor.component.html в папке SportsStore/src/app/admin и включите в него разметку из листинга 9.26.

Листинг 9.26. Файл productEditor.component.html в папке SportsStore/src/app/admin

```
<div class="bg-primary p-a-1" [class.bg-warning]="editing">
    <h5>{{editing  ? "Edit" : "Create"}} Product</h5>
</div>
<form novalidate #form="ngForm" (ngSubmit)="save(form)" >
    <div class="form-group">
        <label>Name</label>
        <input class="form-control" name="name" [(ngModel)]="product.name" />
    </div>
```

```
    <div class="form-group">
        <label>Category</label>
        <input class="form-control" name="category" [(ngModel)]="product.
                    category" />
    </div>
    <div class="form-group">
        <label>Description</label>
        <textarea class="form-control" name="description"
                  [(ngModel)]="product.description">
        </textarea>
    </div>
    <div class="form-group">
        <label>Price</label>
        <input class="form-control" name="price" [(ngModel)]="product.price" />
    </div>
    <button type="submit" class="btn btn-primary" [class.btn-warning]="editing">
        {{editing ? "Save" : "Create"}}
    </button>
    <button type="reset" class="btn btn-secondary" routerLink="/admin/main/
                products">
        Cancel
    </button>
</form>
```

Шаблон содержит форму с полями для свойств, определяемых классом модели `Product`, кроме свойства `id`, значение которого присваивается автоматически REST-совместимой веб-службой.

Элементы формы изменяют свой внешний вид для того, чтобы различать функции редактирования и создания. Чтобы увидеть, как работают компоненты, проведите аутентификацию для получения доступа к функциям администрирования и щелкните на кнопке Create New Product под таблицей товаров. Заполните форму и щелкните на кнопке Create; новый товар отправляется REST-совместимой веб-службе, где ему присваивается свойство `id`, а сам товар отображается в таблице товаров (рис. 9.5).

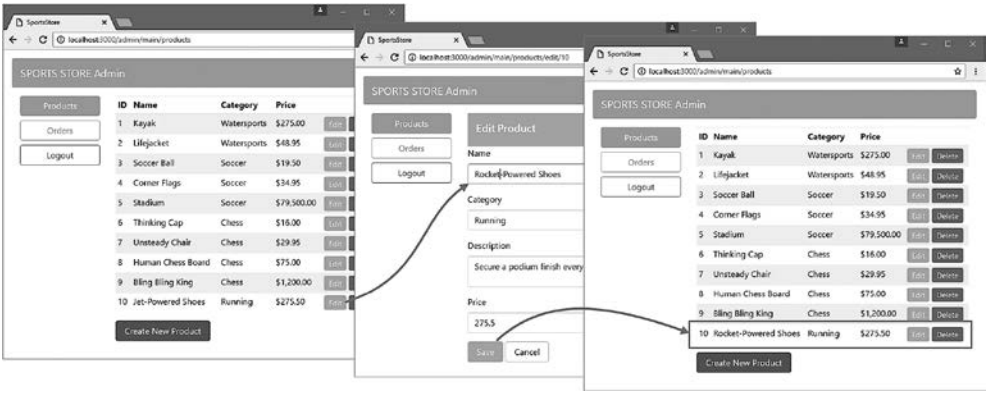


Рис. 9.5. Создание нового товара

Процесс редактирования работает аналогично. Щелкните на одной из кнопок Edit, чтобы просмотреть текущее описание, отредактируйте данные в полях формы и щелкните на кнопке Save для сохранения изменений (рис. 9.6).

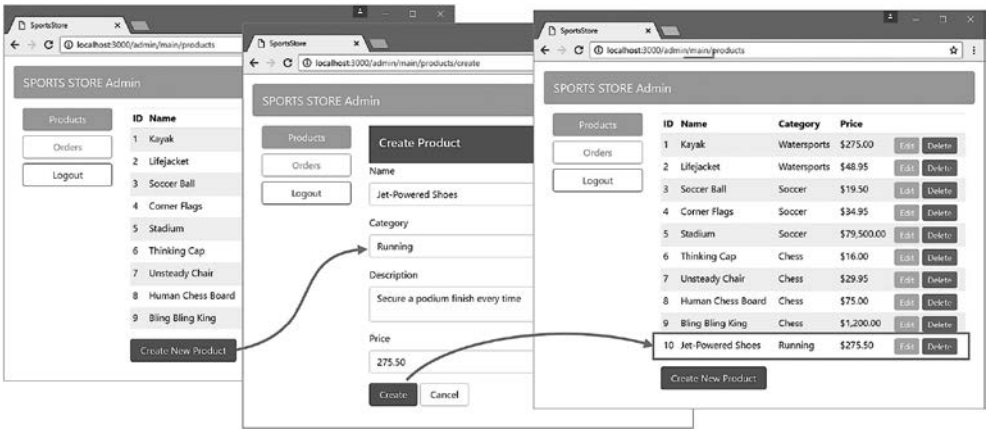


Рис. 9.6. Редактирование существующего товара

Реализация управления заказами

Функция управления заказами проста и незамысловата. В ней используется таблица с набором заказов и кнопками, которые задают свойству `shipped` значение `true` или полностью удаляют заказ. В листинге 9.27 временный контент в компоненте заменяется логикой, необходимой для поддержки этих операций.

Листинг 9.27. Добавление операций в файл orderTable.component.ts

```
import { Component } from "@angular/core";
import { Order } from "../model/order.model";
import { OrderRepository } from "../model/order.repository";

@Component({
    moduleId: module.id,
    templateUrl: "orderTable.component.html"
})
export class OrderTableComponent {
    includeShipped = false;

    constructor(private repository: OrderRepository) {}

    getOrders(): Order[] {
        return this.repository.getOrders()
            .filter(o => this.includeShipped || !o.shipped);
    }

    markShipped(order: Order) {
        order.shipped = true;
```

```
        this.repository.updateOrder(order);
    }

    delete(id: number) {
        this.repository.deleteOrder(id);
    }
}
```

Кроме методов для пометки отправленных заказов и удаления заказов, компонент определяет метод `getOrders` для включения или исключения отправленных заказов в зависимости от значения свойства `includeShipped`. Это свойство используется в шаблоне; создайте файл orderTable.component.html в папке SportsStore/src/app/admin и включите в него разметку из листинга 9.28.

Листинг 9.28. Содержимое файла orderTable.component.html в папке SportsStore/src/app/admin

```
<div class="form-check">
    <label class="form-check-label">
    <input type="checkbox" class="form-check-input" [(ngModel)]="includeShipped"/>
        Display Shipped Orders
    </label>
</div>
<table class="table table-sm">
    <thead>
        <tr><th>Name</th><th>Zip</th><th colspan="2">Cart</th><th></th></tr>
    </thead>
    <tbody>
        <tr *ngIf="getOrders().length == 0">
            <td colspan="5">There are no orders</td>
        </tr>
        <ng-template ngFor let-o [ngForOf]="getOrders()">
            <tr>
                <td>{{o.name}}</td><td>{{o.zip}}</td>
                <th>Product</th><th>Quantity</th>
                <td>
                    <button class="btn btn-warning" (click)="markShipped(o)">
                        Ship
                    </button>
                    <button class="btn btn-danger" (click)="delete(o.id)">
                        Delete
                    </button>
                </td>
            </tr>
            <tr *ngFor="let line of o.cart.lines">
                <td colspan="2"></td>
                <td>{{line.product.name}}</td>
                <td>{{line.quantity}}</td>
            </tr>
        </ng-template>
    </tbody>
</table>
```

Помните, что данные, предоставляемые REST-совместимой веб-службой, сбрасываются при запуске процесса; это означает, что для создания заказов придется использовать корзину и процесс оформления. Когда это будет сделано, вы сможете просматривать заказы и управлять ими в разделе Orders административного раздела (рис. 9.7).

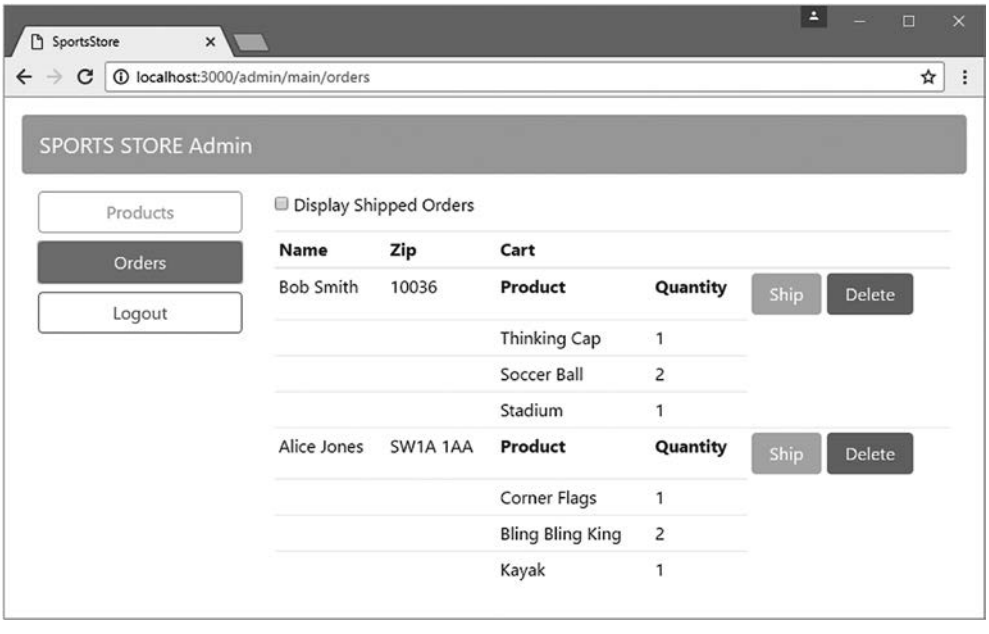


Рис. 9.7. Управление заказами

Итоги

В этой главе мы создали динамически загружаемый функциональный модуль Angular, который содержит административные инструменты для управления каталогом товаров и обработки заказов. В следующей главе приложение SportsStore будет подготовлено к развертыванию.

10 SportsStore: развертывание

В этой главе мы подготовим приложение SportsStore к развертыванию; для этого весь контент приложения и фреймворк Angular будут встроены в один файл, а приложение будет «упаковано» с использованием Docker.

Подготовка приложения к развертыванию

В первоначальной версии этой главы я продемонстрировал, как подготовить приложение к развертыванию вручную. Это был сложный процесс, требовавший тщательной настройки. Пакет `angular-cli` упростил процесс подготовки до одной команды.

Убедитесь в том, что в системе не работает команда `ng serve` или REST-совместимая веб-служба из предыдущей главы, и выполните следующую команду из папки SportsStore:

```
ng build --target=production
```

Команда приказывает `angular-cli` откомпилировать код приложения и сгенерировать набор оптимизированных файлов JavaScript, содержащих функциональность приложения и всю функциональность Angular, от которой оно зависит. Процесс построения может занять некоторое время; после его завершения будет создана папка SportsStore/dist. Кроме файлов JavaScript, присутствует файл index.html, скопированный из папки SportsStore/src и модифицированный для использования вновь построенных файлов.

Контейнеризация приложения SportsStore

В конце этой главы мы создадим для приложения SportsStore контейнер, который заменяет инструменты разработки, использовавшиеся в предыдущих главах, и обозначает переход от разработки к развертыванию. В следующих разделах мы создадим контейнер Docker. На момент написания книги технология Docker была самым популярным способом контейнеризации — создания усеченной версии Linux с минимумом функциональности, необходимой для запуска приложения. Docker поддерживается большинством облачных платформ и систем хостинга, а инструментарий работает во всех основных операционных системах.

Установка Docker

Начните с загрузки и установки инструментария Docker на машине разработки (*www.docker.com/products/docker*). Существуют версии для macOS, Windows и Linux, а также специализированные версии для облачных платформ Amazon и Microsoft. Для этой главы достаточно бесплатной версии Community Edition.

ВНИМАНИЕ

Одна из проблем с использованием Docker заключается в том, что компания-разработчик время от времени вносит критические изменения. Это может означать, что в будущих версиях приведенный ниже пример может работать не так, как предполагалось. В примерах этой главы используется версия 17.03. Если у вас возникнут проблемы, проверьте репозиторий на наличие обновлений (подробности приведены на странице этой книги на сайте apress.com) или свяжитесь со мной по адресу adam@adam-freeman.com.

Подготовка приложения

Начните с создания файла конфигурации для NPM, который будет использоваться для загрузки дополнительных пакетов, необходимых приложению для использования с контейнером. Создайте файл с именем deploy-package.json в папке SportsStore и включите в нее содержимое листинга 10.1.

Листинг 10.1. Содержимое файла deploy-package.json в папке SportsStore

```
{
  "dependencies": {
    "bootstrap": "4.0.0-alpha.4",
    "font-awesome": "4.7.0"
  },

  "devDependencies": {
    "express": "4.14.0",
    "concurrently": "2.2.0",
    "json-server": "0.8.21",
    "jsonwebtoken": "7.1.9"
  },

  "scripts": {
    "start": "concurrently \"npm run express\" \"npm run json\" ",
    "express": "node server.js",
    "json": "json-server data.js -m authMiddleware.js -p 3500"
  }
}
```

Из секции `dependencies` исключены библиотеки Angular и другие пакеты, включенные в файл `package.json` пакетом `angular-cli` при создании проекта. Это связано с тем, что процесс построения интегрирует весь код JavaScript, необходимый для приложения, в файлы в папке dist; следовательно, в списке должны присут-

ствовать только реально используемые дополнения. В секцию `devDependencies` включены два новых инструмента, которые не использовались при разработке (табл. 10.1).

Таблица 10.1. Новые пакеты NPM, используемые в файле deploy-package.json

Имя	Описание
express	Популярный сервер HTTP, широко используемый как сам по себе, так и многими другими средствами разработчика
concurrently	Простой пакет, позволяющий запустить несколько пакетов NPM из одной команды

Секция `scripts` файла deploy-package.json настроена так, чтобы команда npm start запускала REST-совместимую веб-службу и сервер HTTP, который будет доставлять контент HTML, CSS и JavaScript браузеру.

Настройка сервера HTTP

В листинге 10.1 упоминается пакет `express` — популярный сервер HTTP. Он будет использоваться для обработки запросов HTTP приложением после развертывания.

ПРИМЕЧАНИЕ

Пакет express — превосходный сервер (а в книге используется лишь часть его возможностей), но существует много других вариантов, и подойдет любой веб-сервер. Я выбрал express, потому что он близок к инструментам разработки, использованным в предыдущих главах, и потому что его удобно использовать в контейнерах Docker.

Чтобы настроить `express` для запуска приложения, создайте файл с именем deploy-server.js в папке SportsStore и добавьте код из листинга 10.2.

Листинг 10.2. Содержимое файла deploy-server.js в папке SportsStore

```
var express = require("express");

var app = express();

app.use("/node_modules",
    express.static("/usr/src/sportsstore/node_modules"));
app.use("/", express.static("/usr/src/sportsstore/app"));

app.listen(3000, function () {
    console.log("HTTP Server running on port 3000");
});
```

Код создает сервер HTTP на порте 3000 и предоставляет файлы из папок /usr/src/sportsstore/app и /usr/src/sportsstore/node_modules.

Создание контейнера

Чтобы создать контейнер, создайте файл с именем Dockerfile (без расширения) в папке SportsStore/deploy и добавьте команды из листинга 10.3.

Листинг 10.3. Содержимое файла Dockerfile в папке SportsStore

```
FROM node:6.9.1

RUN mkdir -p /usr/src/sportsstore

COPY dist /usr/src/sportsstore/app

COPY authMiddleware.js /usr/src/sportsstore/
COPY data.js /usr/src/sportsstore/
COPY deploy-server.js /usr/src/sportsstore/server.js
COPY deploy-package.json /usr/src/sportsstore/package.json

WORKDIR /usr/src/sportsstore

RUN npm install

EXPOSE 3000
EXPOSE 3500

CMD ["npm", "start"]
```

В содержимом Dockerfile используется базовый образ, который был настроен с Node.js. Команды копируют файлы, необходимые для запуска приложения, включая пакет приложения и файл package.json, который будет использоваться для установки пакетов, необходимых для запуска развернутого приложения.

Выполните следующую команду из папки SportsStore, чтобы создать образ, содержащий приложение SportsStore вместе со всеми необходимыми инструментами и пакетами:

```
docker build . -t sportsstore -f Dockerfile
```

В процессе обработки инструкций из файла Docker выполняется загрузка и установка пакетов NPM, а файлы конфигурации и файлы с кодом копируются в образ.

Запуск приложения

После того как контейнер будет создан, запустите его следующей командой:

```
docker run -p 3000:3000 -p3500:3500 sportsstore
```

Чтобы протестировать приложение, введите в браузере адрес *http://localhost:3000*. На экране появляется ответ, предоставленный веб-сервером, выполняемым в контейнере (рис. 10.1).

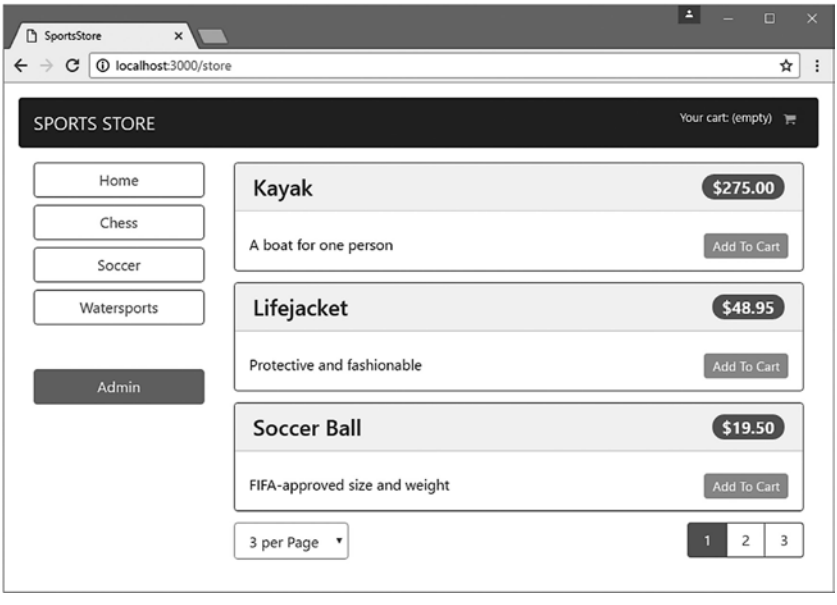


Рис. 10.1. Запуск приложения SportsStore в контейнере

Чтобы остановить контейнер, введите следующую команду:

```
docker ps
```

На экране появляется список работающих контейнеров (некоторые поля опущены для краткости):

```
CONTAINER ID        IMAGE               COMMAND             CREATED
ecc84f7245d6        sportsstore         "npm start"         33 seconds ago
```

Введите следующую команду с указанием идентификатора контейнера:

```
docker stop ecc84f7245d6
```

Итоги

В этой главе мы завершили работу над приложением SportsStore. Вы узнали, как подготовить приложение Angular к развертыванию и как легко преобразовать приложение Angular в контейнер — например, в контейнер Docker.

На этом закончены вступительные главы. Далее мы перейдем к углубленному изучению материала, и вы узнаете, как работают возможности, использованные при создании приложения SportsStore.

11 Создание проекта Angular

В этой главе процесс создания нового проекта Angular и приложений будет рассмотрен более подробно. Я объясню смысл каждого программного пакета, каждого конфигурационного файла и каждого файла с программным кодом. В результате будет создано простое приложение Angular, которое, откровенно говоря, делает не так уж много. Однако к концу главы вы поймете, как части проекта взаимодействуют друг с другом, и заложите основу для применения расширенных возможностей, описанных в следующих главах. В табл. 11.1 приведена краткая сводка материала главы.

Таблица 11.1. Сводка материала главы

Проблема	Решение	Листинг
Подготовка документа HTML для приложения Angular	Добавьте элемент с именем app	1
Добавление пакетов, необходимых для работы приложения и процесса разработки	Создайте файл package.json и используйте команду npm install	2, 3
Передача аннотаций типов компилятору TypeScript	Используйте команду typings для загрузки и установки информации типов	4
Настройка компилятора TypeScript	Создайте файл tsconfig.json	5–7
Обеспечение перезагрузки документа HTML браузером при внесении изменений в проект	Используйте lite-server или другой пакет в качестве сервера HTTP для разработки	8, 9
Создание простого приложения Angular	Создайте модель данных, корневой компонент, корневой модуль и файл начальной загрузки	10–16
Обеспечение разрешимости зависимостей модулей в приложении	Настройка загрузчика модулей JavaScript	17, 18

Подготовка проекта Angular с использованием TypeScript

Работа над новым проектом Angular начинается с настройки процесса разработки. В предыдущих главах вы просто создавали файлы с заданным содержимым, но в этой главе я опишу весь процесс шаг за шагом и подробно объясню, что для этого потребуется.

Прежде чем приступать к работе, необходимо выбрать редактор кода, установить новую версию браузера и новую версию Node.js. Если вы еще не занимались этим, обратитесь к главе 2 за инструкциями.

ПРИМЕЧАНИЕ

Чтобы создать проект, не обязательно выполнять каждый шаг в этой главе. Прилагаемый к книге архив примеров (его можно загрузить на сайте apress.com) содержит все проекты всех глав. В нем также имеется проект, который я создал для этой главы.

Создание структуры папок проекта

Простейшая структура проекта Angular состоит из двух папок. Папка верхнего уровня содержит все файлы проекта, а папка app содержит файлы с кодом Angular. Для этой главы я создал папку верхнего уровня example, а в ней — папку app. Краткие описания этих папок приведены в табл. 11.2.

Таблица 11.2. Папки, необходимые для проекта

Имя	Описание
example	Корневая папка со всеми файлами проекта, включая статический контент (графика, файлы HTML и т. д.) и конфигурационные файлы, описанные далее в этой главе
example/app	Файлы с кодом Angular

Создание документа HTML

Работа Angular зависит от документа HTML, который сообщает браузеру, какие файлы JavaScript необходимы для загрузки и запуска приложения. Когда вы начинаете работу над новым проектом Angular, одним из первых шагов становится создание файла HTML и настройка механизма для передачи его браузеру, что подразумевает установку веб-сервера.

По умолчанию документу HTML присваивается имя index.html. Этот документ создается в корневой папке проекта. Создайте файл HTML и включите в него контент из листинга 11.1.

Листинг 11.1. Содержимое файла index.html в папке example

```
<!DOCTYPE html>
<html>
<head>
    <title>Angular</title>
    <meta charset="utf-8" />
</head>
<body>
    <app>This is static content</app>
</body>
</html>
```

После того как структурные блоки приложения и средства разработки будут готовы, элемент app будет использоваться для вывода динамического контента. Впрочем, пока это всего лишь статический временный заполнитель.

Подготовка конфигурации проекта

Проекты Angular зависят от ряда программных пакетов. Самый простой способ загрузки и установки таких пакетов использует программу NPM (Node Package Manager), описанную в главе 2. Для подготовки пакетов, которые будут добавлены позднее в этой главе, создайте файл package.json в папке example и добавьте в него контент из листинга 11.2.

Листинг 11.2. Содержимое файла package.json в папке example

```
{
  "dependencies": {
  },
  "devDependencies": {
  },
  "scripts": {
  }
}
```

NPM использует имя package.json для своих конфигурационных файлов. Файл делится на три секции (см. листинг 11.2), которые будут заполняться по мере настройки и подготовки проекта. Все три секции описаны в табл. 11.3.

Таблица 11.3. Секции в файле package.json

Имя	Описание
dependencies	Пакеты, необходимые для работы приложения (включая Angular)
devDependencies	Пакеты, используемые в процессе разработки (например, компиляторы и тестовые фреймворки)
scripts	Команды, используемые для компиляции, тестирования и запуска приложения

Добавление пакетов

На следующем шаге в проект добавляются пакеты, необходимые для поддержки разработки и запуска приложения (листинг 11.3). В списке указывается имя каждого пакета и необходимая версия.

ВНИМАНИЕ

Для получения правильных результатов очень важно использовать точные номера версий, приведенные в листинге (и в других листингах). Даже если вы используете в своих проектах более поздние версии, для примеров книги выберите пакеты из листинга 11.3.

Листинг 11.3. Добавление пакетов в файл package.json

```
{
  "dependencies": {
    "@angular/common": "2.2.0",
    "@angular/compiler": "2.2.0",
    "@angular/core": "2.2.0",
    "@angular/platform-browser": "2.2.0",
    "@angular/platform-browser-dynamic": "2.2.0",
    "@angular/upgrade": "2.2.0",
    "reflect-metadata": "0.1.8",
    "rxjs": "5.0.0-beta.12",
    "zone.js": "0.6.26",
    "core-js": "2.4.1",
    "classlist.js": "1.1.20150312",
    "systemjs": "0.19.40",
    "bootstrap": "4.0.0-alpha.4"
  },

  "devDependencies": {
      "lite-server": "2.2.2",
      "typescript": "2.0.2",
      "typings": "1.3.2",
      "concurrently": "2.2.0"
  },

  "scripts": {
    "start": "concurrently \"npm run tscwatch\" \"npm run lite\" ",
    "tsc": "tsc",
    "tscwatch": "tsc -w",
    "lite": "lite-server",
    "typings": "typings",
    "postinstall": "typings install"
  }
}
```

Конкретные пакеты описаны ниже, но они предоставляют базовую функциональность, необходимую для начала разработки проектов Angular. После того как вы отредактируете файл package.json, выполните следующую команду из папки example:

```
npm install
```

NPM загружает пакеты, указанные в листинге 11.3, и устанавливает их в папку с именем node_modules. Процесс требует времени, потому что каждый пакет зависит от других пакетов, многие из которых обладают собственными зависимостями. Когда установка будет завершена, NPM выводит список установленных пакетов. В процессе установки выводятся различные сообщения об ошибках и предупреждения; отчасти это объясняется тем, что NPM вообще выдает много информации, а отчасти тем, что конфигурация из листинга 11.3 зависит от других возможностей, которые будут настроены позднее в этой главе. Когда на экране появится длинный список установленных пакетов, все готово.

ПОЧЕМУ МЫ ИСПОЛЬЗУЕМ NPM

Я использую NPM, потому что этот менеджер стал стандартным механизмом управления пакетами JavaScript. Мир разработки JavaScript разбит на великое множество малых, специализированных пакетов, которые объединяются для реализации богатой, гибкой функциональности. Чтобы вручную загрузить пакеты, необходимые для проекта, вам придется обработать длинный список зависимостей; каждую зависимость в этом списке необходимо загрузить, установить и проанализировать ее собственные зависимости. Даже для такой простой конфигурации, как в листинге 11.3, граф зависимостей содержит более 400 пакетов. Управлять пакетами вручную трудно, поэтому программа NPM заслуживает внимания.

Главная проблема при использовании NPM — большое количество зависимостей в проекте. Зависимости между пакетами могут задаваться неформально, что дает NPM определенную гибкость в выборе версии. Это обстоятельство может вызвать проблемы при публикации обновленных версий пакетов, в которых были обнаружены ошибки. Когда вы обновляете установленные пакеты или перемещаете проект на новую машину, набор пакетов слегка изменяется, что может привести к нарушению работоспособности приложения. Такое случалось со мной во время работы над книгой: неожиданная «поломка» примера в действительности была обусловлена обновлением пакета, спрятанного где-то глубоко в дереве зависимостей. Обычно такие проблемы быстро решаются разработчиками пакетов, но они все равно нарушают процесс разработки (и их трудно найти).

Если у вас возникнут проблемы с примерами, попробуйте использовать проекты из архива, прилагаемого к книге (*apress.com*). Я воспользовался командой npm shrinkwrap для построения файла со списком версий всех пакетов, использованных при написании книги; используйте те же версии при загрузке проектов и выполнении команды npm install.

Пакеты в секции dependencies

Записи в секции `dependencies` в листинге 11.3 добавляют в проект Angular вместе с пакетами, от которых зависит работа Angular, а также рядом дополнительных пакетов, задействованных во многих веб-приложениях; эти пакеты перечислены в табл. 11.4.

ПРИМЕЧАНИЕ

Некоторые пакеты, использованные в книге, находятся в состоянии предварительной версии. В большинстве случаев (как для пакета rxjs) это объясняется тем, что предварительная версия использовалась командой Angular в ходе тестирования перед выпуском. Исключение составляет пакет bootstrap, который я выбрал для оформления контента в приложении. Как упоминалось в предыдущем разделе, вам стоит использовать те же предварительные версии для получения ожидаемых результатов в примерах, даже если в своих проектах вы будете использовать более поздние версии.

Таблица 11.4. Пакеты в секции dependencies

Имя	Описание
@angular/*	Пакеты предоставляют функциональность Angular, разбитую на несколько модулей. Каждый модуль предоставляет свой набор возможностей; модули из листинга 11.3 образуют хорошую основу для новых веб-приложений. В следующих главах будут добавлены другие модули для поддержки конкретных возможностей, например работы с формами HTML
reflect-metadata	Пакет реализует API для рефлексии, используемый при анализе декораторов классов
rxjs	Пакет Reactive Extensions используется для реализации системы обнаружения изменений Angular в привязках данных (см. главы 12–17), а также непосредственно некоторыми возможностями Angular (главы 23–27)
zone.js	Пакет Zone.js предоставляет контекст выполнения для асинхронных задач и используется для вычисления шаблонных выражений
core-js	Пакет Core-JS предоставляет поддержку новых возможностей JavaScript для браузеров, в которых они не поддерживаются напрямую. За дополнительной информацией обращайтесь по адресу https://github.com/zloirock/core-js
classlist.js	Пакет предоставляет недостающие возможности для IE9
systemjs	Загрузчик модулей (см. раздел «Настройка загрузчика модулей JavaScript»)
bootstrap	Bootstrap — фреймворк HTML, который используется для стилевого оформления контента HTML в книге. За дополнительной информацией обращайтесь по адресу http://getbootstrap.com

Пакеты в секции devDependencies

В секции `devDependencies` перечислены инструменты разработки, которые будут использоваться для создания приложений Angular (табл. 11.5). Большинству таких пакетов требуются конфигурационные файлы, которые будут описаны после таблицы.

Таблица 11.5. Пакеты в секции devDependencies

Имя	Описание
lite-server	Этот пакет предоставляет сервер HTTP для разработки, используемый в книге. За дополнительной информацией обращайтесь к разделу «Настройка сервера HTTP для разработки»
typescript	Пакет обеспечивает поддержку языка TypeScript, включая компилятор. За дополнительной информацией обращайтесь к разделу «Настройка TypeScript»
typings	Пакет предоставляет информацию типов для популярных пакетов JavaScript, чтобы с ними было удобнее работать в TypeScript. За дополнительной информацией обращайтесь к разделу «Настройка TypeScript»
concurrently	Пакет позволяет выполнять несколько команд npm одновременно. За дополнительной информацией обращайтесь к разделу «Запуск процессов-наблюдателей»

Настройка TypeScript

TypeScript использует файлы определения типов для удобства работы с библиотеками, написанными на «простом» JavaScript. Эти определения типов размещаются по адресу *http://definitelytyped.org*, а для управления ими используется пакет `typings`, добавленный в файл package.json в листинге 11.3. Выполните следующие команды из папки example, чтобы загрузить и установить определения типов, которые понадобятся в этой части книги:

```
npm run typings -- install dt~core-js --save --global
npm run typings -- install dt~node --save --global
```

Эти команды соответствуют записи `typings` в секции `scripts` файла package.json; пакет `typings` используется для получения определений типов из репозитория Definitively Typed (*http://definitelytyped.org*) для пакетов `core-js` и `node`. Аргумент `--global` сообщает `typings`, что определения типов применяются к проекту в целом, а аргумент `--save` создает конфигурационный файл, который может использоваться для простой повторной установки информации типов (эта возможность может быть полезна при извлечении проекта из системы управления версиями новым разработчиком). После выполнения команды в папке `example` появляются папки typings/globals/core-js и typings/global/node, содержащие определения типов, и файл typings.json с данными о загруженной информации типов (листинг 11.4). В вашем файле typings.json могут содержаться другие номера версий.

Листинг 11.4. Содержимое файла typings.json в папке example

```
{
  "globalDependencies": {
    "core-js": "registry:dt/core-js#0.0.0+20160914114559",
    "node": "registry:dt/node#6.0.0+20161110151007"
  }
}
```

РАБОТА С ОПРЕДЕЛЕНИЯМИ ТИПОВ

Если вам понадобится определение типов для пакета, обратитесь к списку на сайте definitelytyped.org или поищите информацию при помощи программы typings. Например, следующая команда ищет определения типов для пакета core-js:

```
npm run typings -- search core-js
```

Два дефиса (--) приказывают npm передать следующие аргументы программе typings. Аргумент search сообщает typings, что вы ищете информацию типов, а аргумент core-js задает имя пакета. Результат выполнения команды выглядит примерно так:

```
NAME          SOURCE HOMEPAGE                                DESCRIPTION VERSIONS
core-js dt    https://github.com/zloirock/core-js/           1
```

Чтобы загрузить и установить определение типов, объедините значение SOURCE с NAME, разделив их тильдой (символ ~):

```
npm run typings -- install dt~core-js --save --global
```

Аргумент --save создает файл typings.json с информацией типов; это означает, что в будущем вы сможете загрузить и установить то же определение типов следующей командой:

```
npm run typings install
```

Запись postinstall в секции scripts файла package.json обеспечивает загрузку и установку информации типов после выполнения команды npm install.

Настройка компилятора TypeScript

Компилятору TypeScript необходим конфигурационный файл, который подготовит его к использованию в проекте Angular. Создайте файл tsconfig.json в папке example и добавьте в него контент из листинга 11.5.

Листинг 11.5. Содержимое файла tsconfig.json в папке example

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "moduleResolution": "node",
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true
  },
  "exclude": [ "node_modules" ]
}
```

Компиляцией кода TypeScript управляют многочисленные параметры конфигурации. Приведенные в листинге параметры образуют базовый набор, обязательный для любого приложения Angular; они описаны в табл. 11.6.

Таблица 11.6. Параметры компилятора TypeScript в приложениях Angular

Имя	Описание
target	Целевая версия JavaScript для компилятора. Компилятор преобразует TypeScript в простой код JavaScript, использующий только возможности заданной версии. Значение es5 соответствует стандарту ES5, поддерживаемому большинством браузеров, включая старые версии; оно распространяется на большинство браузеров, поддерживаемых Angular
module	Формат создания модулей JavaScript; значение должно соответствовать загрузчику, используемому в проекте. В листинге задано значение commonjs; создаваемые модули могут использоваться с многими разными загрузчиками модулей JavaScript, включая загрузчик SystemJS, описанный в разделе «Настройка загрузчика модулей JavaScript»
moduleResolution	Режим обработки команд import компилятором. Со значением node пакеты ищутся в папке node_modules, где их размещает NPM
emitDecoratorMetadata	Со значением true компилятор включает информацию о декораторе, к которой можно обратиться при помощи пакета reflect-metadata. Это необходимо для приложений Angular
experimentalDecorators	Параметр необходим для emitDecoratorMetadata
exclude	Параметр сообщает компилятору, какие каталоги следует игнорировать

Чтобы проверить, как работает компилятор, добавьте файл compilertest.ts в папку example и включите в него код из листинга 11.6.

Листинг 11.6. Содержимое файла compilertest.ts в папке example

```
export class Test {
    constructor(public message: string) {}
}
```

В листинге определяется простой класс с одним свойством `message`, определяемым через конструктор.

Чтобы протестировать компилятор, выполните следующую команду из папки `example`:

```
npm run tsc
```

Эта команда соответствует записи `tsc` в секции `scripts`. Компилятор находит все файлы JavaScript в проекте, компилирует их и при необходимости сообщает обо всех обнаруженных ошибках. Тестовый класс должен компилироваться без проблем, а в результате должен быть сгенерирован файл compilertest.js в папке example. Содержимое этого файла приведено в листинге 11.7.

Листинг 11.7. Содержимое файла compilertest.js в папке example

```
"use strict";
var Test = (function () {
    function Test(message) {
        this.message = message;
    }
    return Test;
}());
exports.Test = Test;
```

TypeScript преобразует класс в простой код JavaScript и упаковывает его в формат модуля, заданный параметром конфигурации `module` из листинга 11.5.

Настройка сервера HTTP для разработки

Приложения Angular передаются браузеру веб-сервером. В этой книге я использую серверный пакет lite-server; он основан на популярном пакете `BrowserSync`. Он предоставляет простой сервер HTTP, который поддерживает обнаружение изменений в файлах и перезагрузку браузера.

ВНИМАНИЕ

Не используйте lite-server при реальной эксплуатации приложений. Есть много веб-серверов, подходящих для рабочих условий, включая сервер express, на котором базируется lite-server; я использовал его в главе 10 для подготовки приложения SportsStore при развертывании. За информацией о самых популярных вариантах обращайтесь по адресу en.wikipedia.org/wiki/Web_server#Market_share.

Запустите сервер HTTP, выполнив следующую команду из папки example:

```
npm run lite
```

Команда использует запись `lite`, определенную в секции `scripts` листинга 11.3, для запуска сервера HTTP. На экране появляется новое окно (или вкладка браузера), как показано на рис. 11.1, а в нем открывается URL по умолчанию *http://localhost:3000*.

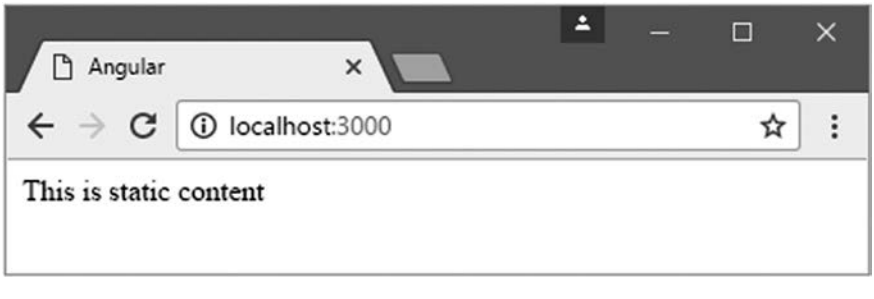


Рис. 11.1. Содержимое исходного проекта

Щелкните правой кнопкой мыши в окне браузера и выберите команду View Source или View HTML; вы увидите, что в документ HTML был добавлен элемент `script`:

```
<!DOCTYPE html>
<html>
<head>
    <title>Angular</title>
    <meta charset="utf-8" />
</head>
<body>
    <script id="__bs_script__">//<![CDATA[
        document.write("<script async src='/browser-sync/browser-sync-
        client.2.16.0.js'><\/script>".replace("HOST", location.hostname));//]]>
    </script>
    <app>This is static content</app>
</body>
</html>
```

Элемент `script` добавляется пакетом `BrowserSync`; он сохраняет открытое подключение к серверу. Когда какой-либо из файлов проекта будет изменен, сервер отправляет сигнал, заставляющий браузер обновить контент. Чтобы убедиться в этом, внесите в файл index.html изменения, показанные в листинге 11.8.

Листинг 11.8. Добавление контента в файл index.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Angular</title>
    <meta charset="utf-8" />
</head>
<body>
    <app>
        This is static content
        <input />
    </app>
</body>
</html>
```

Браузер перезагружает документ HTML сразу же после сохранения изменений. BrowserSync не просто перезагружает контент при изменении; также обеспечивается синхронизация нескольких браузеров, так что прокрутка документа или щелчок на элементе управления в одном браузере приводит к применению того же действия во всех остальных браузерах, в которых отображается тот же документ.

Чтобы увидеть, как работает синхронизация, откройте другое окно браузера или запустите другой браузер и откройте URL по умолчанию *http://localhost:3000*. Весь текст, вводимый в текстовом поле в одном браузере, также появляется во втором браузере (рис. 11.2).

Синхронизация удобна, но она может создать проблемы в приложениях Angular, которые не всегда реагируют ожидаемым образом на события, генерируемые BrowserSync для представления действий пользователя в других окнах. Чтобы изменить конфигурацию, создайте файл bs-config.js в папке example и добавьте код конфигурации из листинга 11.9.

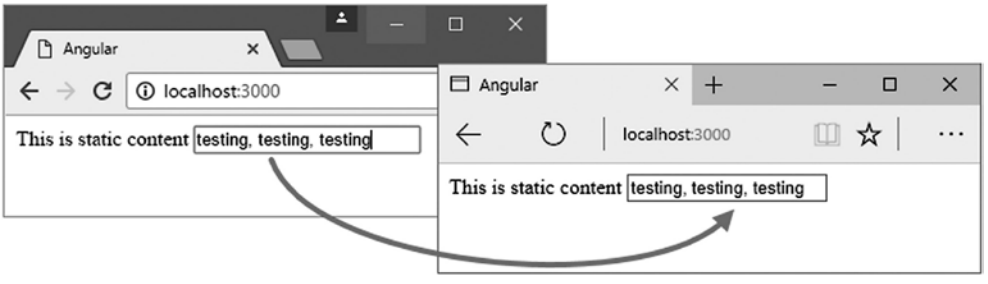


Рис. 11.2. Синхронизация нескольких окон браузера

Листинг 11.9. Содержимое файла bs-config.js в папке example

```
module.exports = {
    ghostMode: false,
    reloadDelay: 1000,
    reloadDebounce: 1000,
    injectChanges: false,
    minify: false
}
```

Эти свойства конфигурации описаны в табл. 11.7.

ПРИМЕЧАНИЕ

BrowserSync обладает широким набором функций, которые могут настраиваться в файле bs-config.js. За дополнительной информацией обращайтесь по адресу https://www.browsersync.io/docs/options.

Таблица 11.7. Параметры конфигурации BrowserSync для приложений Angular

Имя	Описание
ghostMode	Параметр управляет синхронизацией событий между браузерами (рис. 11.2). Значение false отключает синхронизацию
reloadDelay	Задержка сообщения об обновлении, отправляемого браузерам при обнаружении изменений в файлах. Помогает предотвратить повторные перезагрузки при записи нескольких файлов JavaScript компилятором TypeScript
reloadDebounce	Параметр запрещает BrowserSync отправлять сообщения о перезагрузке браузерам до истечения заданного периода времени
injectChanges	Если этот параметр равен false, BrowserSync не будет пытаться внедрять некоторые изменения в существующий документ вместо того, чтобы перезагружать документ
minify	Значение false запрещает BrowserSync проводить минимизацию файлов JavaScript

Запуск процессов-наблюдателей

Пакет `concurrently` добавлен в проект для того, чтобы NPM мог запускать сразу несколько задач. Чтобы организовать эффективный рабочий процесс, компилятор TypeScript должен обнаруживать изменения в файлах с кодом и компилировать их, после чего сервер разработки должен обнаруживать новые файлы JavaScript и обновлять браузер. Эту задачу решает запись `start` в секции `scripts` файла package.json. Завершите все работающие процессы из предыдущих разделов и выполните следующую команду из папки example:

```
npm start
```

Компилятор TypeScript запускается в режиме наблюдения, чтобы он компилировал новые или изменившиеся файлы TypeScript. Генерируемые файлы активизируют проверку изменений BrowserSync и заставляют браузер перезагрузить файл HTML.

ПРИМЕЧАНИЕ

Удобно, когда компилятор TypeScript и сервер HTTP работают в одном процессе, но это усложняет обнаружение ошибок компилятора TypeScript в потоке сообщений о запросах файлов у сервера HTTP. Вы также можете открыть два окна командной строки и запустить эти программы отдельно: команда npm run tscwatch в одном терминале запускает компилятор, а команда npm run lite в другом терминале запускает веб-сервер.

Начало разработки приложений Angular с TypeScript

Структура приложения Angular может показаться довольно запутанной, особенно на первых порах. Чтобы вы лучше поняли, как разные фрагменты связаны друг с другом, на рис. 11.3 изображены основные структурные блоки, встречающиеся в приложениях Angular.

Количество структурных блоков на первый взгляд пугает, но вскоре вы поймете, что делает каждый блок и как они сочетаются друг с другом. Ниже я добавлю в проект некоторые фундаментальные структурные блоки и объясню, что они собой представляют и как работают. В дальнейших главах будет приведена более подробная информация, а также рассмотрены структурные блоки, которые не используются в проектах начального уровня. Чтобы вам было проще сориентироваться, в табл. 11.8 приведены краткие описания всех структурных блоков на рис. 11.3.

ПРИМЕЧАНИЕ

Не беспокойтесь, если описания в таблице покажутся непонятными: понимание роли и взаимодействий между блоками — один из важных этапов изучения Angular. К концу главы вы будете в общих чертах понимать, как устроены приложения Angular, а к концу книги разберетесь в происходящем во всех подробностях.

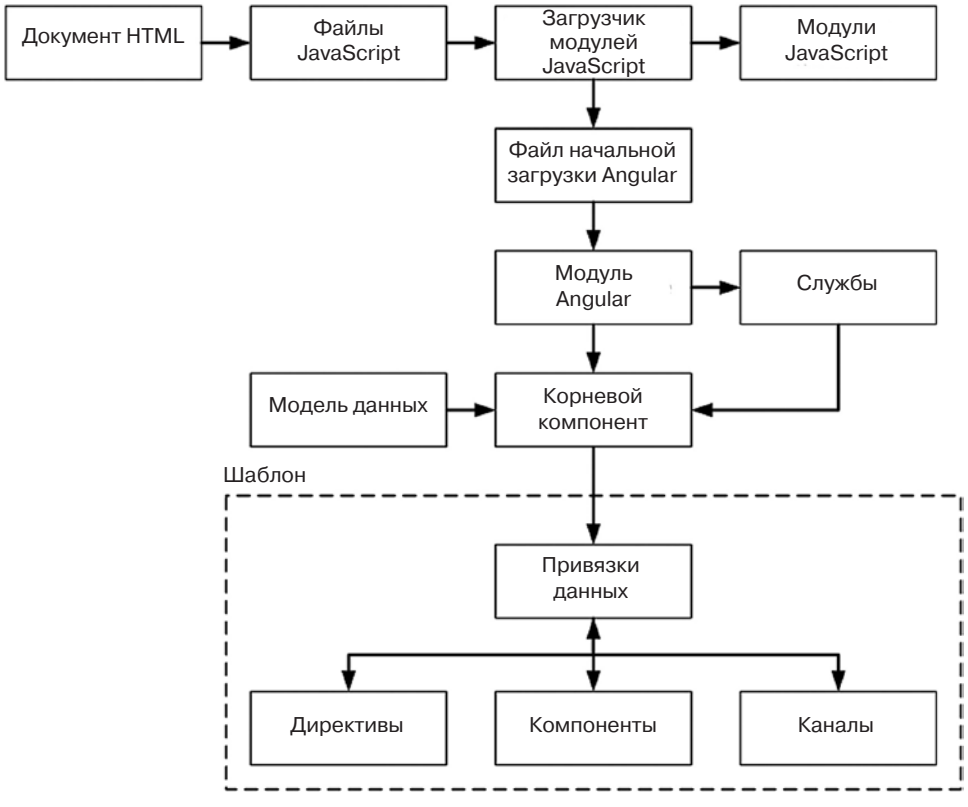


Рис. 11.3. Структурные блоки базового приложения Angular

Таблица 11.8. Структурные блоки приложения Angular

Имя	Описание
Документ HTML	Документ HTML, который запрашивается браузером и содержит весь контент, необходимый для загрузки и запуска приложения Angular. Я создал документ HTML для проекта в листинге 11.1
Файлы JavaScript	Обычные файлы JavaScript, добавляемые в документ HTML с использованием элементов script. Файлы JavaScript используются для добавления недостающих возможностей, которые необходимы для Angular, но не поддерживаются браузером и загрузчиком модулей JavaScript. Список необходимых файлов JavaScript приведен в разделе «Обновление документа HTML»
Загрузчик модулей	Компонент JavaScript, отвечающий за загрузку модулей Angular и других структурных элементов приложения. За дополнительной информацией обращайтесь к разделам «Настройка загрузчика модулей JavaScript» и «Применение загрузчика модулей JavaScript»

Таблица 11.8 (*окончание*)

Имя	Описание
Модули JavaScript	Файлы JavaScript, упакованные в определенный формат, который позволяет загрузчику модулей управлять зависимостями между пакетами для обеспечения правильности загрузки кода JavaScript приложения и его выполнения в правильном порядке. Функциональность фреймворка Angular предоставляется в виде набора модулей JavaScript и некоторых необходимых библиотек поддержки
Файл начальной загрузки Angular	Файл TypeScript, который настраивает Angular и задает модуль Angular, загружаемый для запуска приложения. За дополнительной информацией обращайтесь к разделу «Начальная загрузка приложения»
Модуль Angular	У каждого проекта Angular имеется корневой модуль, который содержит описание приложения, а также может содержать дополнительные модули для группировки взаимосвязанных функций (такая структура упрощает управление приложением). Базовый модуль этой главы приведен в разделе «Создание модуля Angular», а подробное описание модулей приводится в главе 21
Модель данных	Модель предоставляет данные приложения и логику работы с ними. За дополнительной информацией обращайтесь к разделу «Создание модели»
Корневой компонент	Корневой компонент является точкой входа приложения. Он отвечает за генерирование динамического контента, отображаемого для пользователя. За дополнительной информацией обращайтесь к разделу «Создание шаблона и корневого компонента». Компоненты подробно описаны в главе 17
Шаблон	Контент HTML, отображаемый корневым компонентом. За дополнительной информацией обращайтесь к разделу «Создание шаблона и корневого компонента»
Привязки данных	Аннотации в шаблоне, которые сообщают Angular, как вставлять динамический контент (например, значения данных) и как реагировать на взаимодействия с пользователем. Привязки данных описаны в главе 12
Директивы	Классы, преобразующие элементы HTML для генерирования динамического контента. За дополнительной информацией обращайтесь к главе 13
Компоненты	Компоненты добавляют в приложение контент, предоставляющий функциональность приложения (см. главу 17)
Каналы	Классы, используемые для форматирования значений данных, отображаемых для пользователя (см. главу 18)

Запомнить все файлы в проекте Angular непросто, поэтому в табл. 11.9 перечислены файлы и папки, которые будут созданы далее, а также указываются структурные блоки из табл. 11.8, которым они соответствуют. Я добавлю другие файлы в следующих главах, когда мы будем рассматривать работу структурных блоков более подробно.

Таблица 11.9. Файлы и папки в проекте example

Имя	Описание
example/app/main.ts	Файл содержит код начальной загрузки, который предоставляет Angular имя модуля Angular. См. раздел «Начальная загрузка приложения»
example/app/app.module.ts	Файл содержит модуль Angular, предоставляющий имя корневого компонента. См. раздел «Создание модуля Angular»
example/app/product.model.ts	Файл является частью модели данных. См. раздел «Создание модели данных»
example/app/datasource.model.ts	Файл является частью модели данных. См. раздел «Создание модели данных»
example/app/repository.model.ts	Файл является частью модели данных. См. раздел «Создание модели данных»
example/app/template.html	Файл содержит шаблон — комбинацию элементов HTML и динамического контента. См. раздел «Создание шаблона и корневого компонента»
example/app/component.ts	Файл содержит корневой (и только корневой) компонент, содержащий данные и логику поддержки шаблона. См. раздел «Создание шаблона и корневого компонента»
example/index.html	Файл содержит документ HTML. См. раздел «Обновление документа HTML»
example/package.json	Файл содержит список пакетов, используемых приложением или задействованных в процессе разработки. Он также содержит команды, которые могут выполняться из командной строки
example/typings.json	Файл содержит список описаний типов, которые будут добавляться в проект для компилятора TypeScript. См. раздел «Настройка TypeScript»
example/tsconfig.json	Файл содержит конфигурацию компилятора TypeScript. См. раздел «Настройка TypeScript»
example/bs-config.js	Файл содержит конфигурацию сервера HTTP для разработки. См. раздел «Настройка сервера HTTP для разработки»
example/systemjs.config.js	Файл содержит конфигурацию загрузчика модулей JavaScript. См. раздел «Настройка загрузчика модулей JavaScript»
example/typings	Папка содержит информацию типов для компилятора TypeScript
example/node_modules	В папке хранятся пакеты, указанные в пакете package.json

Создание модели данных

Модель данных в приложениях Angular открывает доступ к данным приложения и средствам для работы с ними. Из всех структурных блоков приложений Angular модель оказывается наименее регламентированной. Во всех остальных местах Angular требует применения определенных декораторов или использования определенных частей API, но к модели предъявляется всего одно требование: предоставление доступа к данным, необходимым приложению; все подробности относительно того, как это должно делаться и как будут выглядеть данные, остаются на усмотрение разработчика.

Поначалу это выглядит немного странно, и, возможно, вам будет трудно понять, с чего следует начать, но по сути модель можно разделить на три части:

- класс, который описывает данные модели;
- источник данных, который загружает и сохраняет данные (обычно на сервере);
- репозиторий, который обеспечивает выполнение операций с данными модели.

В следующих разделах мы создадим простую модель, которая предоставляет функциональность, необходимую для описания возможностей Angular в следующих главах.

Создание описательного класса модели

Описательные классы, как следует из названия, описывают данные в приложении. В реальном проекте обычно существует множество классов, которые в полной мере описывают данные, с которыми работает приложение, но чтобы начать работу с этой главой, мы создадим один простой класс. Создайте файл с именем product.model.ts в папке app и определите класс из листинга 11.10.

Листинг 11.10. Содержимое файла product.model.ts в папке app

```
export class Product {

    constructor(public id?: number,
                public name?: string,
                public category?: string,
                public price?: number) {}
}
```

Класс `Product` определяет свойства для идентификатора товара, названия товара, категории и цены. Свойства определяются как необязательные аргументы конструктора, что может пригодиться, когда мы займемся созданием объекта `Product` с использованием формы HTML (см. главу 14).

Создание источника данных

Источник данных предоставляет данные приложению. Самый распространенный тип источника данных использует протокол HTTP для запроса данных у сервера (см. главу 24). В этой главе нам понадобится нечто попроще — источник данных, который сбрасывался в известное состояние при каждом запуске приложения,

чтобы примеры гарантированно возвращали ожидаемый результат. Создайте файл datasource.model.ts в папке app и добавьте в него код из листинга 11.11.

Листинг 11.11. Содержимое файла datasource.model.ts в папке example

```
import {Product} from "./product.model";

export class SimpleDataSource {
    private data:Product[];

    constructor() {
        this.data = new Array<Product>(
        new Product(1, "Kayak", "Watersports", 275),
        new Product(2, "Lifejacket", "Watersports", 48.95),
        new Product(3, "Soccer Ball", "Soccer", 19.50),
        new Product(4, "Corner Flags", "Soccer", 34.95),
        new Product(5, "Thinking Cap", "Chess", 16));
    }

    getData(): Product[] {
        return this.data;
    }
}
```

Данные в этом классе жестко зафиксированы в коде; это означает, что любые изменения, внесенные в приложение, будут потеряны при перезагрузке браузера. В реальных приложениях такая ситуация неприемлема, но она идеальна для примеров книги.

Создание репозитория модели

Последним шагом при построении простой модели становится определение репозитория, который предоставит доступ к данным из источника данных и позволит работать с ними в приложении. Создайте файл repository.model.ts в папке app и определите класс из листинга 11.12.

Листинг 11.12. Содержимое файла repository.model.ts в папке app

```
import { Product } from "./product.model";
import { SimpleDataSource } from "./datasource.model";

export class Model {
    private dataSource: SimpleDataSource;
    private products: Product[];
    private locator = (p:Product, id:number) => p.id == id;

    constructor() {
        this.dataSource = new SimpleDataSource();
        this.products = new Array<Product>();
        this.dataSource.getData().forEach(p => this.products.push(p));
    }

    getProducts(): Product[] {
```

```
        return this.products;
    }

    getProduct(id: number) : Product {
        return this.products.find(p => this.locator(p, id));
    }

    saveProduct(product: Product) {
        if (product.id == 0 || product.id == null) {
            product.id = this.generateID();
            this.products.push(product);
        } else {
            let index = this.products
                .findIndex(p => this.locator(p, product.id));
            this.products.splice(index, 1, product);
        }
    }

    deleteProduct(id: number) {
        let index = this.products.findIndex(p => this.locator(p, id));
        if (index > -1) {
            this.products.splice(index, 1);
        }
    }

    private generateID(): number {
        let candidate = 100;
        while (this.getProduct(candidate) != null) {
            candidate++;
        }
        return candidate;
    }
}
```

Класс `Model` определяет конструктор, который получает исходные данные от класса источника данных и предоставляет доступ к ним через набор методов. Эти методы (их набор типичен для методов, определяемых для репозиториев) перечислены в табл. 11.10.

Таблица 11.10. Методы класса Model

Имя	Описание
getProducts	Метод возвращает массив, содержащий все объекты Product в модели
getProduct	Метод возвращает один объект Product, заданный значением идентификатора
saveProduct	Метод обновляет существующий объект Product или добавляет новый объект в модель
deleteProduct	Метод удаляет из модели объект Product, заданный значением идентификатора

Реализация репозитория может показаться странной, потому что объекты данных хранятся в стандартном массиве JavaScript, но методы, определяемые классом `Model`, представляют данные так, словно они хранятся в коллекции объектов `Product`, индексируемой по свойству `id`. При написании репозитория для данных модели следует учитывать два основных фактора. Во-первых, отображаемые данные должны представляться настолько эффективно, насколько это возможно. В нашем примере это означает, что все данные модели должны представляться в форме с возможностью перебора — например, в массиве. Это важно, потому что перебор может выполняться достаточно часто (см. главу 16). Другие операции класса `Model` не столь эффективны, но они будут использоваться реже.

Во-вторых, необходимо сохранить возможность представления неизмененных данных Angular для работы. В главе 13 я объясню, почему это важно, но в контексте реализации репозитория это означает, что метод `getProducts` при многократном вызове должен возвращать один и тот же объект — если только другой метод или другая часть приложения не изменят данные, которые предоставляет метод `getProducts`. Если метод возвращает разные объекты при каждом вызове (и даже разные массивы, содержащие одинаковые объекты), Angular выдаст сообщение об ошибке. Это может породить непредвиденные проблемы.

Например, в спецификацию ES6 включен класс `Map`, который позволяет хранить объекты с ключом; использование `Map` сделает реализацию методов `getProduct`, `saveProduct` и `deleteProduct` более эффективной по сравнению с листингом 11.12, но при запросе его содержимого будет генерироваться новая серия объектов, что создаст проблемы для Angular. С учетом обоих обстоятельств в реализации репозитория лучше всего хранить данные в массиве и смириться со снижением эффективности некоторых операций.

Создание шаблона и корневого компонента

Шаблон представляет собой фрагмент разметки HTML, который будет отображаться для пользователя. Шаблоны бывают как простыми, так и сложными, от отдельного элемента HTML до сложного блока контента. Чтобы создать шаблон, создайте файл template.html в папке app и добавьте элементы HTML из листинга 11.13.

Листинг 11.13. Содержимое файла template.html в папке app

```
<div class="bg-info p-a-1">
    There are {{model.getProducts().length}} products in the model
</div>
```

Большая часть этого шаблона состоит из стандартной разметки HTML, но часть между двойными фигурными скобками (`{{` и `}}` в элементе `div`) является примером привязки данных. При отображении шаблона Angular обрабатывает контент и обнаруживает привязку. Содержащиеся в привязках выражения вычисляются, а результаты отображаются в документе HTML.

Логика и данные, необходимые для поддержки шаблона, предоставляются компонентом — классом TypeScript, к которому был применен декоратор `@Component`.

Чтобы предоставить компонент для шаблона из листинга 11.13, создайте файл component.ts в папке app и определите класс из листинга 11.14.

Листинг 11.14. Содержимое файла component.ts в папке app

```
import { Component } from "@angular/core";
import { Model } from "./repository.model";

@Component({
    selector: "app",
    templateUrl: "app/template.html"
})
export class ProductComponent {
    model: Model = new Model();
}
```

Декоратор `@Component` настраивает компонент. Свойство `selector` задает элемент HTML, к которому будет применяться директива; он соответствует элементу `app` в документе HTML, созданном в листинге 11.1 (и измененном в листинге 11.8).

```
...
<body>
    <app>This is static content <input /></app>
</body>
...
```

Свойство `templateUrl` в директиве `@Component` задает контент, который будет использоваться для замены элемента `app`; в этом примере свойство задает файл template.html.

Класс компонента, который в данном примере называется `ProductComponent`, отвечает за передачу шаблону данных и логики, необходимых для привязок. В данном случае класс `ProductComponent` определяет одно свойство `model`, которое предоставляет доступ к объекту `Model`.

Создание модуля Angular

В каждом проекте присутствует корневой модуль, который отвечает за описание приложения для Angular. Создайте файл с именем app.module.ts (это имя традиционно используется для корневого модуля) в папке app и добавьте код из листинга 11.15.

Листинг 11.15. Содержимое файла app.module.ts в папке app

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { ProductComponent } from "./component";

@NgModule({
    imports: [BrowserModule],
    declarations: [ProductComponent],
```

```
    bootstrap: [ProductComponent]
})
export class AppModule {}
```

В декораторе `@NdModule` представлены метаданные, которые описывают приложение и предоставляют некоторые критичные функции. Свойство `imports` задает зависимости, используемые приложением, а модуль `BrowserModule`, указанный в листинге 11.15, обеспечивает встроенные функции шаблонов, такие как последовательности `{{` и `}}`, включенные в файл template.html (об этом позднее в этой главе).

Свойство `declarations` описывает функции, предоставляемые приложением для внешнего доступа. Эта функция полезна при создании библиотек функциональности Angular, предназначенных для других приложений (см. главу 21), но в простых приложениях свойство `declarations` используется только для регистрации корневого компонента (такого, как класс `PrpductComponent` из листинга 11.14).

Свойство `bootstrap` сообщает Angular, что класс `ProductComponent` является точкой входа приложения. При запуске приложения Angular использует метод из декоратора `@Component`, примененного к классу `ProductComponent`, для запуска приложения (см. раздел «Запуск приложения» далее в этой главе).

Начальная загрузка приложения

Процесс начальной загрузки сообщает Angular о компоненте, созданном в предыдущем разделе. Он связывает функциональность, предоставляемую фреймворком Angular, с кодом приложения. Начальная загрузка осуществляется определением класса TypeScript; создайте его в файле main.ts в папке app (листинг 11.16).

Листинг 11.16. Содержимое файла main.ts в папке app

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

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

Angular позволяет запускать приложения на разных платформах, каждая из которых предоставляет исполнительную среду для загрузки и запуска приложений. Класс `platformBrowserDynamic` предоставляет исполнительную среду для браузеров, а метод `bootstrapModule` сообщает Angular, какой модуль описывает приложение.

ПРИМЕЧАНИЕ

Обратите внимание: в аргументе модуля bootstrapModule передается имя класса, а не экземпляр этого класса. Другими словами, следует использовать вызов bootstrapModule(AppModule), а не bootstrapModule(new AppModule()) или bootstrapModule(“AppModule”).

Класс `platformBrowserDynamic` включает JIT-компилятор Angular; это означает, что браузер должен обрабатывать файлы JavaScript, которые он получает, при каждой загрузке приложения. В главе 10 было продемонстрировано использование AoT-компилятора, который выполняет этот шаг до развертывания приложения и требует внесения изменений в файле начальной загрузки, чтобы для запуска приложения использовалась другая платформа.

Настройка загрузчика модулей JavaScript

Термином «*модуль*» (module) в приложениях Angular обозначаются два разных понятия. *Модулем Angular* называется класс TypeScript, метаданные которого описывают приложение.

Как упоминалось в главе 6, *модуль JavaScript* представляет собой файл, содержащий код JavaScript. Модули JavaScript могут быть как маленькими, содержащими всего один класс (как в нашем примере), так и очень большими, с множеством классов (сгруппированных тем же способом, который был продемонстрирован в главе 10 при подготовке приложения SportsStore к эксплуатации).

Загрузчик модулей JavaScript отвечает за загрузку всех файлов модулей JavaScript, необходимых браузеру для запуска приложения, и выполнение их в правильном порядке. Задача может показаться тривиальной, но правильная загрузка файлов JavaScript может быть непростым делом, потому что структура зависимостей бывает достаточно сложной (особенно в больших приложениях Angular).

Когда в файле TypeScript встречается команда `import`, компилятор объявляет зависимость в простом файле JavaScript, который сообщает загрузчику модулей, что приложение нуждается в другом модуле JavaScript. Важные команды `import` из файла component.ts:

```
...
import { Component } from "@angular/core";
import { Model } from "./repository.model";
...
```

Во время обработки файла component.ts компилятор TypeScript использует команды `import` для передачи загрузчику модулей информации о зависимостях, которые затем включаются в простой файл JavaScript:

```
...
var core_1 = require("@angular/core");
var repository_model_1 = require("./repository.model");
...
```

Компилятор TypeScript использует метод `require` для разрешения зависимостей, потому что я задал значение `commonjs` свойству `module` в файле tsconfig.json в листинге 11.5. Существует несколько разных форматов модулей, но значение `commonjs` позволяет использовать полезную функциональность Angular для компонентов, которая будет описана в главе 21.

ПРИМЕЧАНИЕ

Важно, чтобы значение параметра module в файле tsconfig.json соответствовало используемому загрузчику модулей. Я указал значение commonjs, чтобы компилятор TypeScript генерировал код JavaScript в модулях, работающих в широком спектре загрузчиков, включая пакет SystemJS. Если вы переключитесь на другой загрузчик модулей, необходимо позаботиться о том, чтобы TypeScript создавал совместимые модули JavaScript.

Загрузчики модулей должны быть настроены таким образом, чтобы они могли разрешать зависимости, объявленные в загружаемых файлах. Чтобы настроить SystemJS, создайте файл с именем systemjs.config.js в папке example и добавьте код, приведенный в листинге 11.17.

Листинг 11.17. Содержимое файла systemjs.config.js в папке example

```
(function(global) {

    var paths = {
        "rxjs/*": "node_modules/rxjs/bundles/Rx.min.js",
        "@angular/*": "node_modules/@angular/*"
    }

    var packages = { "app":  {} };

    var angularModules = ["common", "compiler",
     "core", "platform-browser", "platform-browser-dynamic"];
    angularModules.forEach(function(pkg) {
        packages["@angular/" + pkg] = {
            main: "/bundles/" + pkg + ".umd.min.js"
        };
    });

    System.config({ paths: paths, packages: packages });

})(this);
```

Этот конфигурационный файл содержит код JavaScript (вместо статических данных JSON) и содержит три разных способа разрешения зависимостей модулей JavaScript, применяемые к трем разным частям приложения (см. далее). Такой подход типичен для самых сложных приложений независимо от того, написаны они на Angular или нет, потому что важные пакеты JavaScript распространяются разными способами, которые приходится обрабатывать по-разному.

Разрешение модулей RxJS

Первый пакет, настроенный в листинге 11.17 — `rxjs`, — был добавлен в файл package.json в листинге 11.3. Это пакет Reactive Extensions (RxJS), разработанный в Microsoft, и он используется Angular для распространения уведомлений в приложении при изменении модели данных.

Пакет Reactive Extensions распространяется в виде файла Rx.min.js, находящегося в папке node_modules/rxjs/bundles. Файл содержит несколько модулей, и для разрешения всех зависимостей для всех модулей в одном файле я использую функциональность `paths` SystemJS:

```
...
var paths = {
    "rxjs/*": "node_modules/rxjs/bundles/Rx.min.js",
    "@angular/*": "node_modules/@angular/*"
}
...
```

Каждый раз, когда загрузчик модулей обнаруживает в модуле зависимость, начинающуюся с `rxjs/` (например, `rxjs/Subject`), эта зависимость разрешается с использованием кода в файле Rx.min.js.

ПРИМЕЧАНИЕ

Если вы работаете непосредственно с функциональностью RxJS (такая необходимость возникнет в более сложных приложениях), вам нужно будет создать нестандартный модуль RxJS для своего проекта. Процесс будет продемонстрирован в главе 22.

Разрешение нестандартных модулей приложения

Компилятор TypeScript создает новый модуль для каждого обрабатываемого им файла TypeScript; это означает, что модуль будет создан для каждого класса модели и компонента в приложении.

Загрузчик модулей настраивается для разрешения зависимостей нестандартных модулей в папке app:

```
...
var packages = { "app":  {} };
...
```

Объект `packages` используется для регистрации пакетов, модули которых будут обрабатываться загрузчиком. Этот параметр определяется для того, чтобы мы могли воспользоваться настройками по умолчанию, используемыми SystemJS, чтобы зависимости от таких модулей, как app/component, разрешались загрузкой файла app/component.js с сервера.

Разрешение модулей Angular

Фреймворк Angular распространяется в виде набора модулей JavaScript. Если где-то встречается зависимость от такого модуля, как `@angular/core`, она должна разрешаться загрузкой такого файла, как node_modules/@angular/core/bundles/core.umd.min.js. Функция `paths`, которой я воспользовался для пакета RxJS, в такой ситуации не может использоваться сама по себе, потому что имя модуля дважды встречается в имени пути (а функция `paths` такую возможность не поддерживает).

Настройка загрузчика модулей для обработки модулей JavaScript Angular состоит из двух разных шагов. Первый шаг — передача загрузчику модулей информации о каждом модуле Angular, необходимом приложению.

```
...
var angularModules = ["common", "compiler",
    "core", "platform-browser", "platform-browser-dynamic"];
angularModules.forEach(function(pkg) {
    packages["@angular/" + pkg] = {
        main: "/bundles/" + pkg + ".umd.min.js"
    };
});
...
```

Я начал с пяти модулей, необходимых для базового приложения Angular (другие модули будут добавлены в следующих главах). Для каждого из модулей JavaScript Angular у объекта `packages` определяется новое свойство, которое выглядит так:

```
packages["@angular/core"] = {main: "/bundles/core.umd.min.js"}
```

Свойство `main` применяется для определения файла в пакете, который должен использоваться по умолчанию. В данном случае зависимость от модуля `@angular/core` загрузчик разрешает, загружая файл @angular/core/bundles/core.umd.min.js.

На втором шаге настраивается значение `paths`, описывающее файлы в папке node_modules:

```
...
var paths = {
    "rxjs/*": "node_modules/rxjs/bundles/Rx.umd.min.js",
    "@angular/*": "node_modules/@angular/*"
}
...
```

В сочетании с параметром `packages` загрузчик модулей будет правильно обрабатывать зависимости от модулей Angular.

Применение параметров конфигурации

В завершение процесса объекты `paths` и `packages` передаются методу `System.config`, который предоставляет загрузчику модулей информацию о конфигурации:

```
...
System.config({ paths: paths, packages: packages });
...
```

МЕТОД ПРОБ И ОШИБОК: НАСТРОЙКА ЗАГРУЗЧИКА МОДУЛЕЙ

Возможно, в ходе проработки нужной конфигурации вам придется поэкспериментировать: вы используете средства разработчика F12, смотрите, какие файлы запрашиваются, и изменяете настройки загрузчика модулей, чтобы он знал, где найти эти файлы в проекте. Не все пакеты создают столько неудобств, как Angular, но у многих есть свои странности, связанные с упаковкой и распространением. Популярные загрузчики модулей предоставляют параметры конфигурации, достаточно гибкие для большинства

пакетов. В этом разделе я использовал две возможности, предоставляемые SystemJS; полный список приведен по адресу *http://github.com/systemjs/systemjs*.

Очень важно проверять, какие файлы запрашиваются, при добавлении в проект новых пакетов. В противном случае вы рискуете получить рабочее приложение, которое запрашивает больше файлов, чем нужно, или запрашивает несжатые версии, предназначенные для отладки.

Обновление документа HTML

На последнем шаге следует обновить документ HTML, чтобы он включал пакеты, не поставляемые в виде модулей, и загрузчик модулей. В листинге 11.18 выделены изменения, необходимые для нашего примера.

Листинг 11.18. Добавление пакетов JavaScript и загрузчика модулей в файл index.html

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
    <meta charset="utf-8" />
    <script src="node_modules/classlist.js/classList.min.js"></script>
    <script src="node_modules/core-js/client/shim.min.js"></script>
    <script src="node_modules/zone.js/dist/zone.min.js"></script>
    <script src="node_modules/reflect-metadata/Reflect.js"></script>
    <script src="node_modules/systemjs/dist/system.src.js"></script>
    <script src="systemjs.config.js"></script>
    <script>
      System.import("app/main").catch(function(err){ console.error(err); });
    </script>
    <link href="node_modules/bootstrap/dist/css/bootstrap.min.css"
        rel="stylesheet" />
</head>
<body class="m-a-1">
    <app></app>
</body>
</html>
```

Не все пакеты JavaScript, необходимые для Angular, доставляются или загружаются в виде модулей JavaScript. Первые четыре элемента `script` в листинге относятся к пакетам classList.js, Core-JS, Zone.js и Reflect-Metadata, которые загружаются и выполняются браузером в указанном порядке.

ПРИМЕЧАНИЕ

Самая надежная информация о том, какой файл должен загружаться из пакета, приведена на домашней странице проекта. Например, домашняя страница Core-JS находится по адресу https://github.com/zloirock/core-js; на ней описаны различные варианты использования пакета, включая вариант с файлом client/shim.min.js.

Применение загрузчика модулей JavaScript

Загрузчику модулей SystemJS требуются три разных элемента `script`. Первый элемент `script` загружает библиотеку SystemJS.

```
...
<script src="node_modules/systemjs/dist/system.src.js"></script>
...
```

На следующем шаге загружается конфигурационный файл. Так как этот файл содержит код JavaScript (вместо JSON), для его загрузки и исполнения можно использовать элемент `script`:

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

Последний элемент `script` загружает простой файл JavaScript, созданный компилятором TypeScript на базе файла начальной загрузки Angular из листинга 11.18.

```
...
<script>
    System.import("app/main").catch(function(err){ console.error(err); });
</script>
...
```

Метод `System.import` загружает файл и запускает процесс разрешения его зависимостей. В листинге методу `import` передается аргумент `app/main`, который связывается конфигурацией из файла systemjs.config.js с файлом app/main.js. Все ошибки выводятся на консоль JavaScript браузера.

Стилевое оформление контента

Элемент `link` из листинга 11.18 загружает пакет Bootstrap, который используется в этой книге для стилевого оформления контента HTML в примерах. Bootstrap не имеет прямого отношения к Angular, но может использоваться для стилевого оформления как статического контента в документе HTML, так и динамического контента, созданного на базе шаблонов. Документ HTML содержит простой стиль Bootstrap, примененный к элементу `body`.

```
...
<body class="m-a-1">
...
```

Назначение элементу класса `m-a-1` добавляет поля со всех четырех сторон, создавая интервалы между элементом и окружающим его контентом. Также стили Bootstrap присутствуют в файле template.html из листинга 11.13, где они применяются к элементу `div`.

```
...
<div class="bg-info p-a-1">
...
```

Назначение элементу класса `bg-info` определяет цвет фона элемента; здесь `info` — один из наборов стандартизированных цветов (`info`, `danger`, `success` и т. д.). Назначение элементу класса `p-a-1` добавляет отступы со всех четырех сторон элемента, создавая интервалы между контентом и сторонами элемента.

Запуск приложения

После сохранения изменений в файле index.html в листинге 11.18 все структурные блоки находятся на своих местах и приложение готово к запуску. Браузер загружает документ HTML и обрабатывает его содержимое, включая элементы `script` для загрузчика модулей.

Загрузчик модулей загружает файл app/main.js и начинает разрешать его зависимости — корневой модуль приложения и один из файлов модулей JavaScript фреймворка Angular. Каждый из них обладает собственными зависимостями, и загрузчик модулей обрабатывает всю цепочку, загружая соответствующие файлы с сервера по настройкам в своих конфигурационных файлах.

После того как все файлы будут загружены, загрузчик модулей выполняет файл main.js, который осуществляет начальную загрузку приложения Angular. Файл начальной загрузки сообщает Angular, что корневым модулем приложения является класс `AppModule`, а свойства декоратора `@NgModule` сообщают, что корневым компонентом приложения является класс `ProductComponent`. Angular анализирует декоратор `@Component` класса `ProductComponent` и ищет в документе HTML элемент, соответствующий свойству конфигурации `selector` компонента. Контент, заданный свойством `templateUrl` (содержащийся в файле app/template.html), вставляется в документ HTML как контент элемента `selector`. В ходе обработки шаблона Angular встречает символы `{{` и `}}`, обозначающие привязку данных, и вычисляет содержащееся в них выражение.

```
...
<div class="bg-info p-a-1">
    There are {{model.getProducts().length}} products in the model
</div>
...
```

Значение свойства `model` предоставляется компонентом, который создал объект `Model`. Angular вызывает метод `getProducts` для объекта `Model` и вставляет значение свойства `length` результата в документ HTML. Результат показан на рис. 11.4.

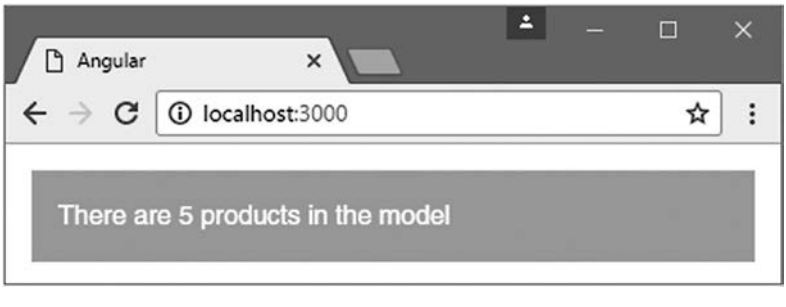


Рис. 11.4. Запуск приложения

Щелкните правой кнопкой мыши на контенте в окне браузера и выберите в контекстном меню команду Inspect или Inspect Element (конкретный интерфейс зависит от браузера; возможно, он будет отображаться только при открытом окне инструментария разработчика F12). Если вы проанализируете модель DOM браузера, которая является динамическим представлением документа HTML, вы увидите, как структурные блоки приложения взаимодействуют друг с другом:

```
<html>
<head>
    <title></title>
    <meta charset="utf-8">
    <script src="node_modules/classlist.js/classList.min.js"></script>
    <script src="node_modules/core-js/client/shim.min.js"></script>
    <script src="node_modules/zone.js/dist/zone.min.js"></script>
    <script src="node_modules/reflect-metadata/Reflect.js"></script>
    <script src="node_modules/systemjs/dist/system.src.js"></script>
    <script src="systemjs.config.js"></script>
    <script>
      System.import("app/main").catch(function(err){ console.error(err); });
    </script>
    <link href="node_modules/bootstrap/dist/css/bootstrap.min.css"
rel="stylesheet">
</head>
<body class="m-a-1"><script id="__bs_script__">//<![CDATA[
    document.write("<script async src='/browser-sync/browser-sync-
    client.js?v=2.16.0'><\/script>".replace("HOST", location.hostname));
    //]]></script><script async="" src="/browser-sync/browser-sync-
    client.js?v=2.16.0"></script>

    <app>
        <div class="bg-info p-a-1">
            There are 5 products in the model
        </div>
    </app>
</body>
</html>
```

Начальный загрузчик, модуль Angular, корневой компонент, шаблон, привязка данных, модель данных и сама среда Angular совместными усилиями генерируют контент, показанный в окне браузера. Возможно, это не выглядит особо выдающимся достижением, потому что приводит к такому простому результату, но создание проекта заложило основу для реализации расширенной — и более интересной — функциональности, описанной в следующих главах.

Итоги

В этой главе мы создали проект разработки Angular и использовали его для знакомства со структурными блоками простого приложения и процесса его создания, компиляции и доставки клиенту. В следующей главе мы займемся углубленным изучением материала, начиная с привязки данных Angular.

12 Привязки данных

В примере из предыдущей главы представлен простой шаблон с привязкой данных, которая отображала количество объектов в модели данных. В этой главе я опишу базовые привязки данных, предоставляемые Angular, и покажу, как они используются при генерировании динамического контента. В следующих главах мы рассмотрим более сложные привязки данных и разберемся, как расширить систему привязки Angular нестандартной функциональностью. В табл. 12.1 представлены привязки данных в контексте.

Таблица 12.1. Привязки данных в контексте

Вопрос	Ответ
Что это такое?	Привязки данных — выражения, которые встраиваются в шаблоны и вычисляются для генерирования динамического контента в документах HTML
Для чего нужны привязки данных?	Привязки данных создают связь между элементами HTML в документах HTML и файлами шаблонов с данными и кодом в приложении
Как они используются?	Привязки данных применяются как атрибуты элементов HTML или специальные последовательности символов в строках
Есть ли у них недостатки или скрытые проблемы?	Привязки данных содержат простые выражения JavaScript, которые вычисляются для генерирования контента. Главная проблема — включение слишком большого объема логики в привязку, потому что такую логику не удастся нормально протестировать или использовать в других местах приложения. Выражения в привязках данных должны быть как можно проще, а сложная логика приложения должна предоставляться компонентами (и другими средствами Angular, такими как каналы)
Есть ли альтернативы?	Нет. Привязки данных — важнейшая часть разработки приложений Angular

В табл. 12.2 приведена краткая сводка материала главы.

Таблица 12.2. Сводка материала главы

Проблема	Решение	Листинг
Динамическое отображение данных в документе HTML	Определите привязку данных	1–4
Настройка элемента HTML	Используйте привязку со стандартным свойством или атрибутом	5, 8
Назначение контента элемента	Используйте привязку со строковой интерполяцией	6, 7
Изменение классов, к которым относится элемент	Используйте привязку классов	9–13
Настройка отдельных стилей, применяемых к элементу	Используйте привязку стилей	14–17
Инициирование обновления модели данных	Используйте консоль JavaScript в браузере	18, 19

Подготовка проекта

В этой главе мы продолжим использовать проект example из главы 11. Чтобы подготовиться к задачам этой главы, я добавил метод в класс компонента (листинг 12.1).

ПРИМЕЧАНИЕ

Если вы не хотите создавать проект «с нуля», загрузите проекты example на сайте apress.com.

Листинг 12.1. Добавление метода в файл component.ts

```
import { Component } from "@angular/core";
import { Model } from "./repository.model";

@Component({
    selector: "app",
    templateUrl: "app/template.html"
})
export class ProductComponent {
    model: Model = new Model();

    getClasses(): string {
        return this.model.getProducts().length == 5 ? "bg-success" : "bg-warning";
    }
}
```

Запустите сервер HTTP для разработки, выполнив следующую команду из папки example:

```
npm start
```

Открывается новое окно браузера, в котором отображается контент на рис. 12.1.

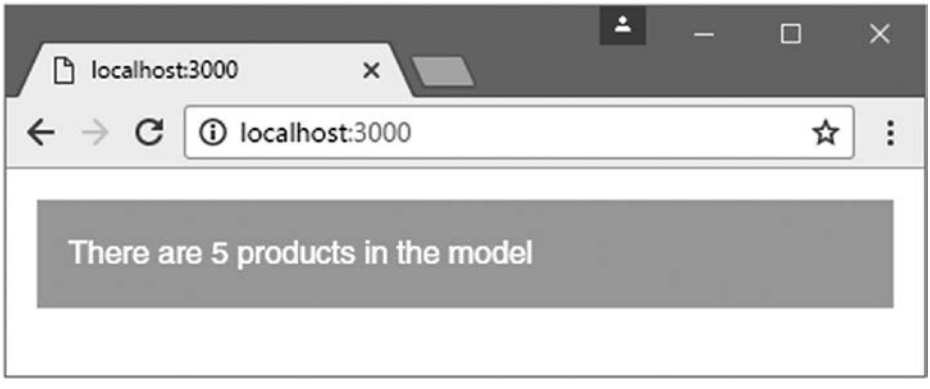


Рис. 12.1. Запуск приложения

Односторонние привязки данных

Односторонние привязки данных используются для генерирования контента для пользователя и являются базовыми структурными блоками для шаблонов Angular. Термин «односторонний» обусловлен тем, что данные передаются в одном направлении; в контексте привязок данных, описанных в этой главе, это означает, что данные передаются от модели данных компоненту, а затем привязке данных для отображения в шаблоне.

Существуют и другие типы привязок данных Angular, которые я опишу в дальнейших главах. Привязки событий работают в другом направлении, от элементов шаблона к остальному коду приложения, и обеспечивают взаимодействие пользователя с приложением. При двусторонних привязках данные могут передаваться в обоих направлениях; эти привязки чаще всего используются в формах. За дополнительной информацией о других привязках обращайтесь к главам 13 и 14.

Чтобы начать работу с односторонними привязками данных, замените контент шаблона в соответствии с листингом 12.2.

Листинг 12.2. Содержимое файла template.html

```
<div [ngClass]="getClasses()" >
    Hello, World.
</div>
```

При сохранении изменений в шаблоне сервер разработки также инициирует обновление браузера. Результат показан на рис. 12.2.

Это простой пример, но он демонстрирует базовую структуру привязки данных, представленную на рис. 12.3.

Привязка данных состоит из четырех частей:

- *Управляющий элемент* — элемент HTML, к которому будет применена привязка (с изменением его внешнего вида, контента или поведения).

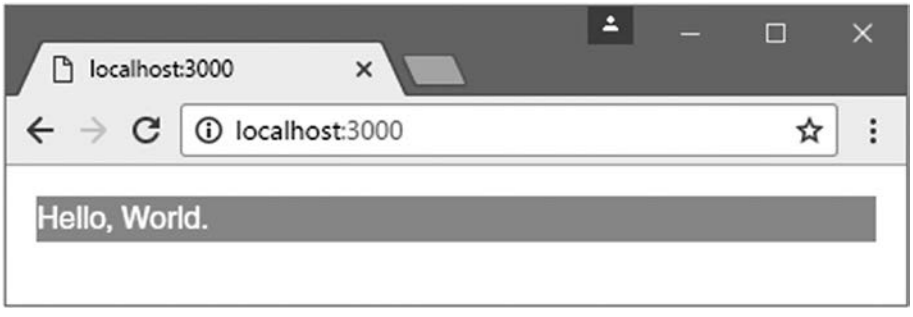


Рис. 12.2. Использование односторонней привязки данных

Рис. 12.3. Строение привязки данных

- *Квадратные скобки* сообщают Angular, что привязка данных является односторонней. Встречая квадратные скобки в привязке данных, Angular вычисляет выражение и передает результат цели привязки, чтобы она могла изменить управляющий элемент.
- *Цель* определяет, что должна делать привязка. Цели делятся на две категории: *директивы* и *привязки свойств*.
- *Выражение* представляет собой фрагмент JavaScript, который вычисляется с использованием компонента шаблона для получения контекста (это означает, что свойства и методы компонента могут включаться в выражение, как метод `getClasses` в приведенном примере).

Взглянув на привязку в листинге 12.2, мы видим, что управляющим элементом является элемент `div`, то есть привязка предназначена для модификации этого элемента. Выражение вызывает метод `getClasses` компонента, определенный в начале главы. Метод возвращает строку с классом Bootstrap, выбранным в зависимости от количества объектов в модели данных.

```
...
getClasses(): string {
    return this.model.getProducts().length == 5 ? "bg-success" : "bg-warning";
}
...
```

Если модель данных содержит не менее пяти объектов, метод возвращает `bg-success` — класс Bootstrap, применяющий зеленый фон. В противном случае метод возвращает `bg-warning` — класс Bootstrap, применяющий желтый фон.

Целью привязки данных в листинге 12.2 является директива — класс, написанный специально для поддержки привязки данных. Angular включает пару полезных встроенных директив, и вы можете создавать собственные директивы для предоставления нестандартной функциональности. Имена встроенных директив начинаются с префикса `ng`, из чего можно сделать вывод, что цель `ngClass` принадлежит к числу таких встроенных директив. Цель обычно сообщает, что делает директива; как можно догадаться по имени, директива `ngClass` добавляет или удаляет управляющий элемент из класса или классов, имена которых возвращаются при вычислении выражения.

Объединяя все сказанное, привязка данных в листинге 12.2 назначает элементу `div` класс `bg-success` или `bg-warning` в зависимости от количества объектов в модели данных. Так как при запуске приложения модель данных содержит пять объектов (потому что исходные данные жестко запрограммированы в классе `SimpleDataSource`, созданном в главе 11), метод `getClasses` возвращает `bg-success` и выдает результат, показанный на рис. 12.3: элемент `div` снабжается зеленым фоном.

Цель привязки

В ходе обработки цели привязки данных Angular сначала проверяет, совпадает ли она с директивой. Большинство приложений использует комбинацию встроенных директив, предоставляемых Angular, и нестандартных директив, предоставляющих специфическую функциональность приложения. Обычно директивы, являющиеся целями привязок данных, можно отличить по содержательному имени, которое дает некоторое представление о том, что делает директива. Встроенные директивы распознаются по префиксу `ng`. Привязка в листинге 12.2 подсказывает, что цель является встроенной директивой, которая имеет отношение к принадлежности управляющего элемента к определенному классу. Для справки в табл. 12.3 приведен список основных встроенных директив Angular с указанием места, где приводится более подробное описание. (Есть и другие директивы, которые описаны в последующих главах, но перечисленные директивы наиболее просты и чаще всего применяются на практике.)

Таблица 12.3. Основные встроенные директивы Angular

Имя	Описание
ngClass	Директива используется для назначения классов управляющим элементам (см. раздел «Назначение классов и стилей»)
ngStyle	Директива используется для назначения отдельных стилей (см. раздел «Назначение классов и стилей»)
ngIf	Директива вставляет контент в документ HTML при условии, что результат вычисления заданного выражения равен true (см. главу 13)

Имя	Описание
ngFor	Директива вставляет контент в документ HTML для каждого элемента в источнике данных (см. главу 13)
ngSwitch ngSwitchCase ngSwitchDefault	Директивы используются для выбора блоков контента, вставляемых в документ HTML, в зависимости от значения выражения (см. главу 13)
ngTemplateOutlet	Директива используется для повторения блока контента (см. главу 13)

Привязки свойств

Если цель привязки не соответствует директиве, то Angular проверяет, может ли цель использоваться для создания привязки свойства. Есть пять разных типов привязок свойств; они перечислены в табл. 12.4 с указанием места, где приводится более подробное описание.

Таблица 12.4. Привязки свойств Angular

Имя	Описание
[свойство]	Стандартная привязка свойства, используемая для задания свойства объекта JavaScript, который представляет управляющий элемент модели DOM (см. раздел «Стандартные привязки свойств и атрибутов»)
[attr.имя]	Привязка атрибута, используемая для задания значений атрибутов управляющих элементов HTML, для которых не существует свойств DOM (см. раздел «Привязки атрибутов»)
[class.имя]	Специальная привязка для настройки принадлежности класса управляющего элемента (см. «Привязки классов»)
[style.имя]	Специальная привязка для настройки стиля управляющего элемента (см. «Привязки стилей»)

Выражения в привязках данных

Выражение в привязке данных представляет собой фрагмент кода JavaScript, результат которого вычисляется для передачи цели значения. В выражении доступны свойства и методы, определяемые компонентом; поэтому привязка в листинге 12.2 может вызвать метод `getClasses` для передачи директиве `ngClass` имени класса, назначаемого управляющему элементу.

Выражения не ограничиваются простым вызовом методов или чтением свойств компонентов; они также могут использоваться для выполнения многих стандартных операций JavaScript. Например, в листинге 12.3 приведено выражение, в котором строковый литерал объединяется с результатом метода `getClasses`.

Листинг 12.3. Выполнение операции JavaScript в выражении из файла template.html

```
<div [ngClass]="'p-a-1 ' + getClasses()" >
    Hello, World.
</div>
```

Выражение заключено в кавычки; это означает, что строковый литерал должен определяться в апострофах. В JavaScript оператором конкатенации является символ `+`, а результатом выражения будет комбинация двух строк:

```
p-a-1 bg-success
```

В результате директива `ngClass` назначает управляющему элементу два класса: `p-a-1` (используется Bootstrap для добавления отступов вокруг контента элемента) и `bg-success` (задает цвет фона).

На рис. 12.4 изображено сочетание этих двух классов.

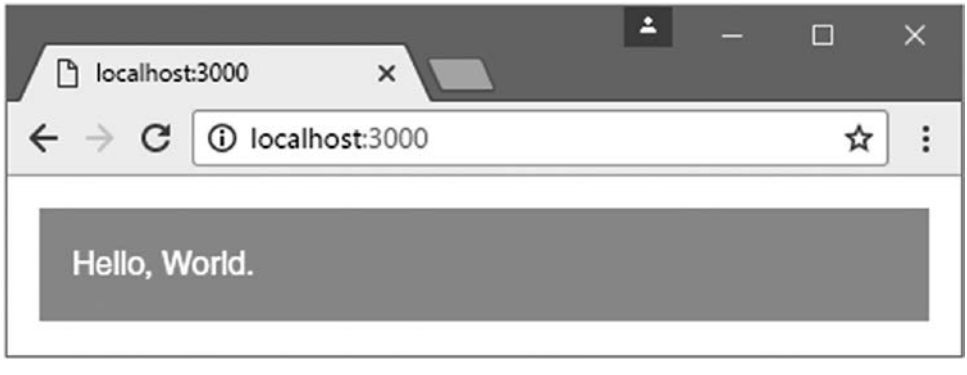


Рис. 12.4. Объединение классов в выражении JavaScript

При написании выражений легко увлечься и включить сложную логику в шаблон. Это может создать проблемы, потому что выражения не проверяются компилятором TypeScript и затрудняют модульное тестирование; это повышает риск того, что ошибки останутся незамеченными до развертывания приложения. Чтобы избежать подобных проблем, используйте максимально простые выражения. В идеале они должны использоваться только для получения данных от компонента и форматирования их для вывода. Вся сложная логика получения и обработки данных должна определяться в компоненте модели, где ее можно откомпилировать и протестировать.

Квадратные скобки

Квадратные скобки (символы `[` и `]`) сообщают Angular о том, что это односторонняя привязка данных с выражением, результат которого необходимо вычислить. Если опустить квадратные скобки, а целью является директива, Angular все равно обработает привязку, но выражение не будет обработано, а символы между апо-

строфами будут переданы директиве как литерал. В листинге 12.4 в шаблон добавляется элемент с привязкой без квадратных скобок.

Листинг 12.4. Привязка данных в template.html без квадратных скобок

```
<div [ngClass]="'p-a-1 ' + getClasses()" >
    Hello, World.
</div>
<div ngClass="'p-a-1 ' + getClasses()" >
    Hello, World.
</div>
```

Если вы проанализируете элемент HTML в режиме просмотра DOM в браузере (щелкните правой кнопкой мыши в окне браузера и выберите в контекстном меню команду Inspect или Inspect Element), то увидите, что его атрибуту `class` был присвоен строковый литерал:

```
<div class="'p-a-1 ' + getClasses()">
```

Браузер пытается обработать классы, назначенные управляющему элементу, но внешний вид элемента не соответствует ожиданиям, потому что имена не соответствуют именам классов, используемым Bootstrap. Это довольно распространенная ошибка; если привязка не дает желаемого эффекта, начните с этой проверки.

Квадратные скобки не единственная разновидность скобок, используемых Angular в привязках данных. В табл. 12.5 перечислены разные конструкции со скобками, смысл каждой из них и даются ссылки на более подробные описания.

Таблица 12.5. Скобки

Имя	Описание
[цель]="выражение"	Квадратные скобки обозначают одностороннюю привязку данных, в которой данные передаются от выражения к цели. Эта глава посвящена различным формам этого типа привязок
{{выражение}}	Привязка со строковой интерполяцией (см. раздел «Привязки со строковой интерполяцией»)
(цель) ="выражение"	Круглые скобки обозначают одностороннюю привязку данных, в которой данные передаются от цели к приемнику, заданному выражением. Такие привязки используются для обработки событий (см. главу 14)
[(цель)] ="выражение"	Такая комбинация скобок обозначает двустороннюю привязку, в которой данные передаются в обоих направлениях между целью и приемником, заданным выражением (см. главу 14)

Управляющий элемент

Управляющий элемент — самая простая часть привязки. Привязки данных могут применяться к любому элементу HTML в шаблоне, и элемент может иметь несколько привязок, каждая из которых управляет своим аспектом внешнего вида

или поведения элемента. Примеры элементов с несколькими привязками будут приведены ниже.

Стандартные привязки свойств и атрибутов

Если цель привязки не соответствует директиве, Angular пытается применить привязку свойства. Далее описаны самые распространенные привязки свойств: стандартная привязка свойства и привязка атрибута.

Стандартные привязки свойств

Браузер использует модель DOM (Document Object Model) для представления документа HTML. Каждый элемент в документе HTML, в том числе и управляющий элемент, представляется объектом JavaScript в DOM. Как и все объекты JavaScript, объекты, представляющие элементы HTML, обладают свойствами. Эти свойства используются для управления состоянием элемента, так что свойство `value`, например, используется для задания содержимого элемента `input`. Когда браузер разбирает документ HTML, при обнаружении каждого нового элемента HTML он создает в DOM объект для его представления и использует атрибуты элемента для задания исходных значений свойств объекта.

Стандартная привязка свойства позволяет задать значение свойства объекта, представляющего управляющий элемент, в соответствии с результатом выражения. Например, задавая цели привязки значение `value`, вы задаете содержимое элемента `input` (листинг 12.5).

Листинг 12.5. Использование стандартной привязки свойства в файле template.html

```
<div [ngClass]="'p-a-1 ' + getClasses()" >
    Hello, World.
</div>
<div class="form-group m-t-1">
    <label>Name:</label>
    <input class="form-control" [value]="model.getProduct(1)?.name || 'None'" />
</div>
```

Новая привязка в этом примере указывает, что свойство `value` должно быть привязано к результату выражения, которое вызывает метод модели данных для получения объекта данных от репозитория по ключу. Может оказаться, что объект данных с таким ключом не существует; в таком случае метод репозитория вернет `null`.

Чтобы защититься от использования `null` для свойства `value` управляющего элемента, привязка использует условный оператор (символ `?`) для успешной передачи результата, возвращаемого методом:

```
...
<input class="form-control" [value]="model.getProduct(1)?.name || 'None'" />
The first product is {{getProduct(1)?.name || 'None'}}.
...
```

Если результат метода `getProduct` отличен от `null`, выражение прочитает значение свойства `name` и использует его как результат. Но если результат метода равен `null`, то свойство `name` прочитано не будет и оператор слияния с `null` (символы `||`) задаст результат `None`.

ЗНАКОМСТВО СО СВОЙСТВАМИ ЭЛЕМЕНТОВ HTML

Чтобы использовать привязки свойств, сначала необходимо понять, какое свойство нужно задать. В спецификации HTML присутствует определенная несогласованность. Имена большинства свойств совпадают с именами атрибутов, задающих их исходное значение; таким образом, если вы привыкли задавать атрибут value для элемента input, то вы сможете добиться того же эффекта, задав свойство value. Некоторые имена свойств не совпадают с именами атрибутов, а некоторые свойства вообще не могут задаваться атрибутами.

Mozilla Foundation предоставляет полезную справочную информацию по всем объектам, которые используются для представления элементов HTML в DOM, по адресу developer.mozilla.org/en-US/docs/Web/API. Для каждого элемента Mozilla предоставляет сводку доступных свойств с описанием их предполагаемого использования. Начните с объекта HTMLElement (developer.mozilla.org/en-US/docs/Web/API/HTMLElement), предоставляющего функциональность, общую для всех элементов. После этого переходите к объектам конкретных элементов (таким, как объект HTMLInputElement, используемый для представления элементов input).

Когда вы сохраняете изменения в шаблоне, браузер обновляется и отображает элемент `input`, контентом которого является свойство `name` объекта данных с ключом 1 в репозитории модели (рис. 12.5).

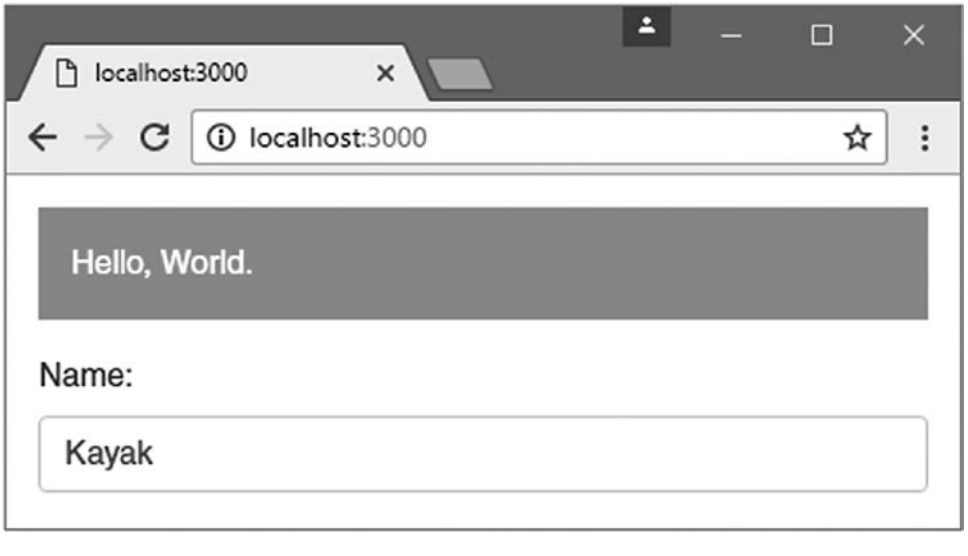


Рис. 12.5. Стандартная привязка свойства

Привязки со строковой интерполяцией

Angular предоставляет особую разновидность стандартных привязок свойств — *привязки со строковой интерполяцией*, используемые для включения результатов выражений в текстовый контекст управляющих элементов. Чтобы понять, почему эта разновидность привязок так полезна, стоит разобраться в том, что необходимо при использовании стандартных привязок свойств.

Свойство `textContent` используется для задания контента элементов HTML; это означает, что контент элемента может задаваться с использованием привязки данных наподобие той, что приведена в листинге 12.6.

Листинг 12.6. Назначение контента элемента в файле template.html

```
<div [ngClass]="'p-a-1 ' + getClasses()"
     [textContent]="'Name: ' + (model.getProduct(1)?.name || 'None')" >
</div>
<div class="form-group m-t-1">
    <label>Name:</label>
    <input class="form-control" [value]="model.getProduct(1)?.name || 'None'" />
</div>
```

Выражение в новой привязке объединяет строковый литерал с результатами вызова метода для задания контента элемента `div`. Выражение в этом примере получилось громоздким; вы должны тщательно следить за кавычками, пробелами и скобками, чтобы в выводе присутствовали ожидаемые результаты. Проблема усугубляется в более сложных привязках, в которых несколько динамических значений чередуются с блоками статического контента. Привязка со строковой интерполяцией упрощает этот процесс: она допускает определение фрагментов выражений в контенте элемента (листинг 12.7).

Листинг 12.7. Использование привязок со строковой интерполяцией в файле template.html

```
<div [ngClass]="'p-a-1 ' + getClasses()">
    Name: {{model.getProduct(1)?.name || 'None'}}
</div>
<div class="form-group m-t-1">
    <label>Name:</label>
    <input class="form-control" [value]="model.getProduct(1)?.name || 'None'" />
</div>
```

Привязка со строковой интерполяцией обозначается парами фигурных скобок (`{{` и `}}`). Один элемент может содержать сразу несколько привязок со строковой интерполяцией.

Angular объединяет контент элемента HTML с содержимым скобок для создания привязки для свойства `textContent`. Результат тот же, что в листинге 12.6 (он показан на рис. 12.6), но процесс написания привязки упрощается, а риск ошибок снижается.

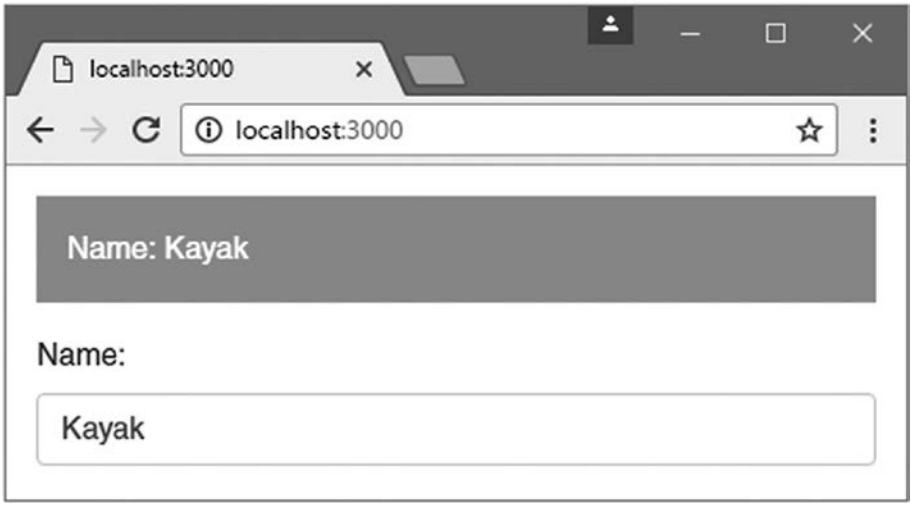


Рис. 12.6. Привязка со строковой интерполяцией

Привязки атрибутов

Из-за странностей спецификаций HTML и DOM не все атрибуты элементов HTML имеют эквивалентные свойства в DOM API, это означает, что некоторые аспекты управляющего элемента не могут быть заданы с использованием стандартной привязки свойств.

Для таких ситуаций Angular предоставляет *привязки атрибутов*, которые используются для задания атрибутов управляющего элемента вместо задания значения объекта JavaScript, представляющего этот элемент в DOM. Самый типичный атрибут, не имеющий соответствующего свойства — `colspan`, — используется для задания числа столбцов, занимаемых элементом `td` в таблице. В листинге 12.8 продемонстрировано использование привязки атрибута для задания элемента `colspan` в зависимости от количества объектов в модели данных.

Листинг 12.8. Использование привязки атрибута в файле template.html

```
<div [ngClass]="'p-a-1 ' + getClasses()">
    Name: {{model.getProduct(1)?.name || 'None'}}
</div>
<div class="form-group m-t-1">
    <label>Name:</label>
    <input class="form-control" [value]="model.getProduct(1)?.name || 'None'" />
</div>
<table class="table table-sm table-bordered table-striped m-t-1">
    <tr>
        <th>1</th><th>2</th><th>3</th><th>4</th><th>5</th>
    </tr>
    <tr>
        <td [attr.colspan]="model.getProducts().length">
```

```
            {{model.getProduct(1)?.name || 'None'}}
        </td>
    </tr>
</table>
```

Привязка атрибута применяется посредством определения цели, у которой имя атрибута снабжается префиксом `attr.` (слово `attr`, за которым следует точка). В листинге привязка атрибута используется для задания значения элемента `colspan` одного из элементов `td` в таблице:

```
...
<td [attr.colspan]="model.getProducts().length">
...
```

Angular вычисляет выражение и задает атрибуту `colspan` значение `result`. Так как модель данных жестко запрограммирована на пять объектов данных, в результате атрибут `colspan` создает ячейку таблицы, занимающую пять столбцов (рис. 12.7).

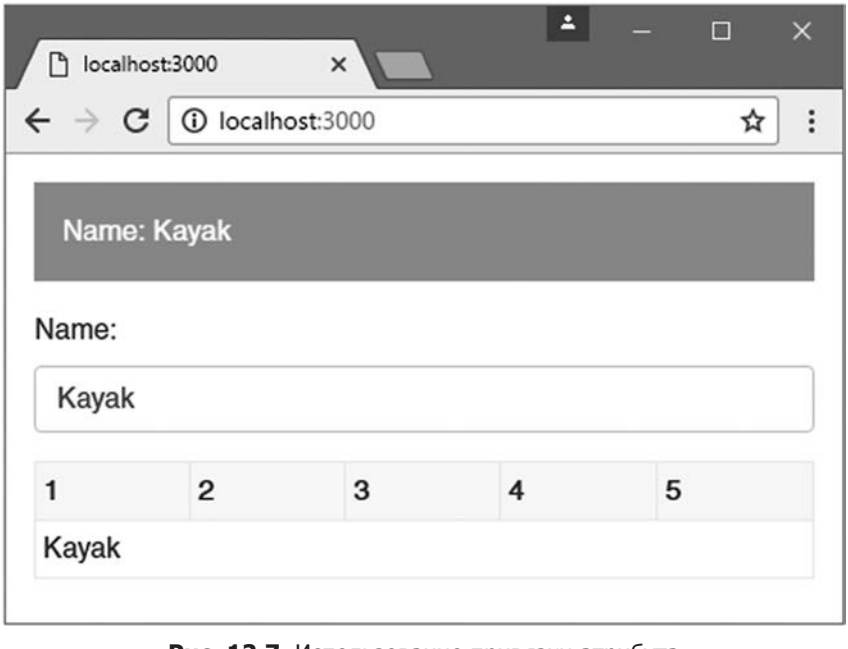


Рис. 12.7. Использование привязки атрибута

Назначение классов и стилей

Angular предоставляет в привязках свойств специальную поддержку для назначения классов управляющим элементам, а также для настройки отдельных свойств стиля. Эти привязки описываются в последующих разделах вместе с подробной информацией о директивах `ngClass` и `ngStyle`, предоставляющих похожую функциональность.

Привязки классов

Существуют три способа использования привязок данных для управления принадлежностью элемента к классам: стандартная привязка свойства, специальная привязка класса и директива `ngClass`. Все три способа описаны в табл. 12.6, они работают по слегка различающимся принципам и применяются в разных обстоятельствах, как описано ниже.

Таблица 12.6. Привязки классов Angular

Пример	Описание
<div [class]="expr">	Привязка вычисляет выражение и использует результат для замены всех существующих назначений классов
<div [class.myClass]="expr">	Привязка вычисляет выражение и использует результат для назначения элементу класса myClass
<div [ngClass]="map">	Привязка назначает элементу несколько классов по данным из объекта map

Назначение всех классов элемента с использованием стандартной привязки

Стандартная привязка свойства может использоваться для назначения всех классов элемента за одну операцию; это может быть полезно, если компонент содержит метод или свойство, возвращающие все классы, к которым должен принадлежать элемент, в виде одной строки, с разделением имен пробелами. В листинге 12.9 показан обновленный метод `getClasses` в компоненте, который возвращает разные строки с именами классов в зависимости от свойства `price` объекта `Product`.

Листинг 12.9. Передача строки с классами в файле component.ts

```
import { Component } from "@angular/core";
import { Model } from "./repository.model";

@Component({
    selector: "app",
    templateUrl: "app/template.html"
})
export class ProductComponent {
    model: Model = new Model();

    getClasses(key: number): string {
        let product = this.model.getProduct(key);
        return "p-a-1 " + (product.price < 50 ? "bg-info" : "bg-warning");
    }
}
```

Результат метода `getClasses` включает класс `p-a-1`, который добавляет отступы вокруг контента управляющего элемента, для всех объектов `Product`. Если значение свойства `price` меньше 50, в результат включается класс `bg-info`, а если 50 и больше — включается класс `bg-warning` (эти классы назначают разные цвета фона).

ПРИМЕЧАНИЕ

Проследите за тем, чтобы имена классов разделялись пробелами.

В листинге 12.10 продемонстрировано применение стандартной привязки свойства в шаблоне для задания свойства `class` управляющих элементов с использованием метода `getClasses` компонента.

Листинг 12.10. Назначение классов с использованием стандартной привязки свойства в файле template.html

```
<div [class]="getClasses(1)">
    The first product is {{model.getProduct(1).name}}.
</div>
<div [class]="getClasses(2)">
    The second product is {{model.getProduct(2).name}}
</div>
```

Когда стандартная привязка свойства используется для задания свойства `class`, результат выражения заменяет все классы, ранее назначенные элементу; это означает, что она может применяться только в том случае, когда выражение привязки возвращает *все* необходимые классы, как в нашем примере. Результат показан на рис. 12.8.

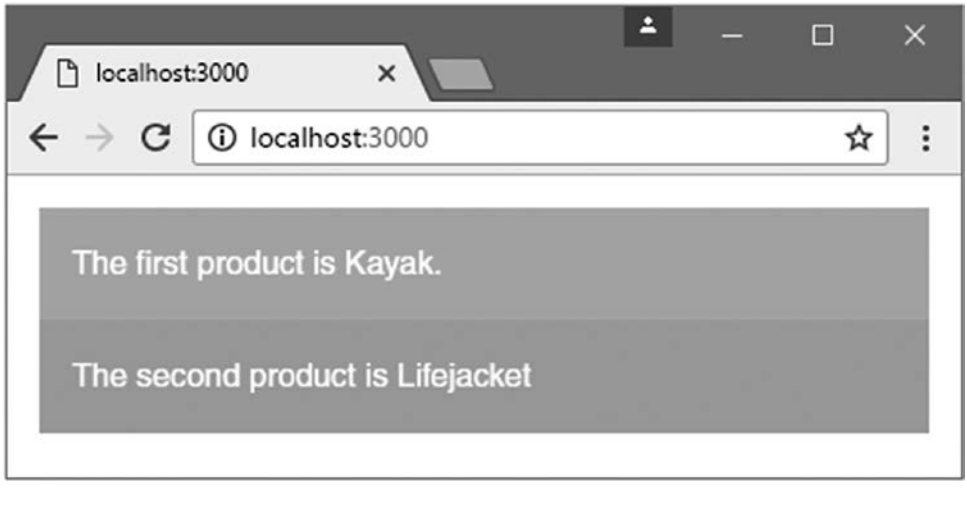


Рис. 12.8. Назначение классов

Назначение отдельных классов с использованием специальной привязки класса

Специальная привязка класса предоставляет более точный механизм управления, чем стандартная привязка свойства. Она позволяет управлять принадлежностью к одному классу с использованием выражения. Например, это может быть полезно, если вы хотите расширить состав классов, к которым принадлежит элемент, вместо полной их замены. В листинге 12.11 продемонстрировано использование специальной привязки класса.

Листинг 12.11. Использование специальной привязки класса в файле template.html

```
<div [class]="getClasses(1)">
    The first product is {{model.getProduct(1).name}}.
</div>
<div class="p-a-1"
     [class.bg-success]="model.getProduct(2).price < 50"
     [class.bg-info]="model.getProduct(2).price >= 50">
        The second product is {{model.getProduct(2).name}}
</div>
```

Специальная привязка класса задается целью, состоящей из слова `class`, точки и имени класса, принадлежностью к которому вы управляете. В листинге задействованы две специальные привязки класса, управляющие принадлежностью к классам `bg-success` и `bg-info`.

Специальная привязка класса добавляет заданный класс к управляющему элементу, если результат выражения является квазиистинным (см. врезку «Квазиистинность и квазиложность»). В данном случае управляющему элементу будет назначен класс `bg-success`, если свойство `price` меньше 50, и класс `bg-info`, если свойство `price` равно 50 и более.

Эти привязки действуют независимо друг от друга и не влияют на классы, к которым уже принадлежит элемент, например класс `p-a-1`, который используется Bootstrap для добавления отступов вокруг контента элемента.

КВАДИИСТИННОСТЬ И КВАЗИЛОЖНОСТЬ

У Javascript есть одна странность: результат выражения может быть квазиистинным (truthy) или квазиложным (falsy), что создает проблемы у несведущих. Следующие результаты всегда являются квазиложными:

– значение false (boolean);

– значение 0 (number);

– пустая строка ("");

– null;

– undefined;

– NaN (специальное значение number).

Все остальные значения являются квазиистинными; это обстоятельство может сбить новичка с толку. Например, значение "false" (строка, символы которой образуют слово false) является квазиистинным. Лучший способ избежать путаницы — использовать только те выражения, при вычислении которых будет получен логический результат true или false.

Назначение классов директивой ngClass

Директива `ngClass` — более гибкая альтернатива для стандартных и специальных привязок свойств. Ее поведение зависит от типа данных, возвращаемого выражением (табл. 12.7).

Таблица 12.7. Типы результата выражения, поддерживаемые директивой ngClass

Имя	Описание
Строка	Управляющему элементу назначаются классы, заданные строкой. Перечисляемые классы разделяются пробелами
Массив	Каждый объект в массиве определяет имя класса, назначаемый управляющему элементу
Объект	Каждое свойство объекта определяет имена одного или нескольких классов, разделенных пробелами. Класс назначается управляющему элементу, если значение свойства является квазиистинным

Поддержка строк и массивов полезна, но главная ценность директивы `ngClass` связана с возможностью использования объекта (называемого *картой*) для определения сложных политик назначения классов. В листинге 12.12 приведен метод компонента, возвращающий объект карты.

Листинг 12.12. Возвращение объекта карты в файле component.ts

```
import { Component } from "@angular/core";
import { Model } from "./repository.model";

@Component({
    selector: "app",
    templateUrl: "app/template.html"
})
export class ProductComponent {
    model: Model = new Model();

    getClasses(key: number): string {
        let product = this.model.getProduct(key);
        return "p-a-1 " + (product.price < 50 ? "bg-info" : "bg-warning");
    }

    getClassMap(key: number): Object {
```

```
        let product = this.model.getProduct(key);
        return {
            "text-xs-center bg-danger": product.name == "Kayak",
            "bg-info": product.price < 50
        };
    }
}
```

Метод `getClassMap` возвращает объект со свойствами, которые представляют имена классов, а значения определяются значениями свойств объекта `Product`, ключ которого передается в методе аргумента. Например, если ключ равен 1, метод возвращает следующий объект:

```
{
  "text-xs-center bg-danger":true,
  "bg-info":false
}
```

Первое свойство назначает управляющему элементу класс `text-xs-center` (используемый Bootstrap для горизонтального выравнивания текста по центру) и класс `bg-danger` (задает цвет фона элемента). Второе свойство дает результат `false`; это означает, что управляющий элемент не будет добавлен в класс `bg-info`. На первый взгляд это выглядит странно: зачем задавать свойство, которое не приводит к назначению класса элементу? Но как вы вскоре увидите, значения выражений автоматически обновляются в соответствии с изменениями в приложении, и возможность определения объекта-карты, который задает принадлежность к классам подобным образом, может быть весьма полезной.

В листинге 12.13 продемонстрировано использование метода `getClassMap` и возвращаемого им объекта в выражении привязки данных с директивой `ngClass`.

Листинг 12.13. Использование директивы ngClass в файле template.html

```
<div class="p-a-1" [ngClass]="getClassMap(1)">
    The first product is {{model.getProduct(1).name}}.
</div>
<div class="p-a-1" [ngClass]="getClassMap(2)">
    The second product is {{model.getProduct(2).name}}.
</div>
<div class="p-a-1" [ngClass]="{'bg-success': model.getProduct(3).price < 50,
                               'bg-info': model.getProduct(3).price >= 50}">
        The third product is {{model.getProduct(3).name}}
</div>
```

Первые два элемента `div` содержат привязки, использующие метод `getClassMap`. Третий элемент `div` демонстрирует альтернативный подход, основанный на определении карты в шаблоне. Для этого элемента принадлежность к классам `bg-info` и `bg-warning` связывается со значением свойства `price` объекта `Product` (рис. 12.9). Будьте осторожны с этим приемом: выражение содержит логику JavaScript, которую довольно трудно протестировать.

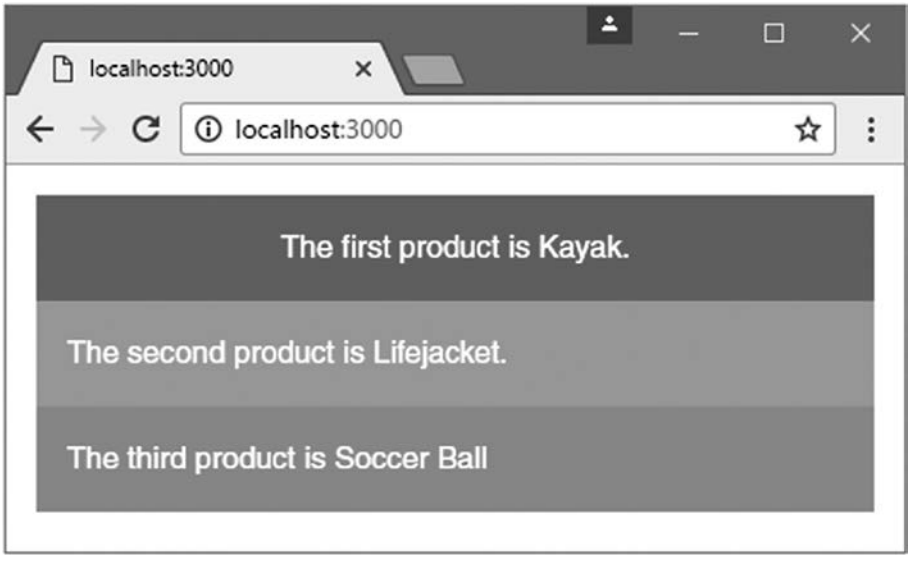


Рис. 12.9. Использование директивы ngClass

Привязки стилей

Существуют три разных способа использования привязок данных для назначения свойства `style` управляющего элемента: стандартная привязка свойства, специальная привязка стиля и директива `ngStyle`. Все три способа описаны в табл. 12.8, а их применение продемонстрировано далее.

Таблица 12.8. Привязки стилей Angular

Пример	Описание
<div [style.myStyle]="expr">	Стандартная привязка свойства, используемая для задания одному стилевому свойству результата выражения
<div [style.myStyle.units]="expr">	Специальная привязка стиля; позволяет задать единицы значения как часть цели
<div [ngStyle]="map">	Привязка задает несколько стилевых свойств в соответствии с данными, содержащимися в объекте

Назначение одного стилевого свойства

Стандартная привязка свойства и специальная привязка стиля используются для задания одного стилевого свойства. Они отличаются тем, что стандартная привязка свойства должна включать единицы, необходимые для задания стиля, а специальная привязка позволяет включить единицы в цель привязки.

Для демонстрации этого различия в листинге 12.14 в компонент добавляются два новых свойства.

Листинг 12.14. Добавление свойств в файл component.ts

```
import { Component } from "@angular/core";
import { Model } from "./repository.model";

@Component({
    selector: "app",
    templateUrl: "app/template.html"
})
export class ProductComponent {
    model: Model = new Model();
    getClasses(key: number): string {
        let product = this.model.getProduct(key);
        return "p-a-1 " + (product.price < 50 ? "bg-info" : "bg-warning");
    }

    getClassMap(key: number): Object {
        let product = this.model.getProduct(key);
        return {
            "text-xs-center bg-danger": product.name == "Kayak",
            "bg-info": product.price < 50
        };
    }

    fontSizeWithUnits: string = "30px";
    fontSizeWithoutUnits: string= "30";
}
```

Свойство `fontSizeWithUnits` возвращает значение с величиной и единицами, в которых эта величина выражена: 30 пикселов. Свойство `fontSizeWithoutUnits` возвращает только величину без информации о единицах. Листинг 12.15 показывает, как эти свойства могут использоваться со стандартными и специальными привязками.

ВНИМАНИЕ

Не пытайтесь использовать стандартную привязку свойства для свойства style, чтобы задать сразу несколько стилевых значений. Объект, возвращаемый свойством style объекта JavaScript, представляющего управляющий элемент в DOM, доступен только для чтения. Некоторые браузеры игнорируют этот факт и допускают изменения, однако результаты непредсказуемы и полагаться на них не следует. Если вы хотите задать несколько стилевых свойств, создайте привязку для каждого из них или воспользуйтесь директивой ngStyle.

Листинг 12.15. Использование привязок стилей в файле template.html

```
<div class="p-a-1 bg-warning">
    The <span [style.fontSize]="fontSizeWithUnits">first</span>
     product is {{model.getProduct(1).name}}.
</div>
```

```
<div class="p-a-1 bg-info">
    The <span [style.fontSize.px]="fontSizeWithoutUnits">second</span>
     product is {{model.getProduct(2).name}}
</div>
```

Цель привязки — `style.fontSize` — задает размер шрифта для контента управляющего элемента. Выражение этой привязки использует свойство `fontSizeWithUnits`, значение которого включает единицы (`px` — пикселы), необходимые для задания размера шрифта.

Цель специальной привязки — `style.fontSize.px` — сообщает Angular о том, что значение выражения задает величину в пикселах. Это позволяет привязке использовать свойство `fontSizeWithoutUnits` компонента, не включающее единицы.

ПРИМЕЧАНИЕ

Стилевые свойства можно задавать в формате имен свойств JavaScript ([style.fontSize]) или в формате имен свойств CSS ([style.font-size]).

Результаты обеих привязок одинаковы: размер шрифта элементов `span` задается равным 30 пикселам (рис. 12.10).

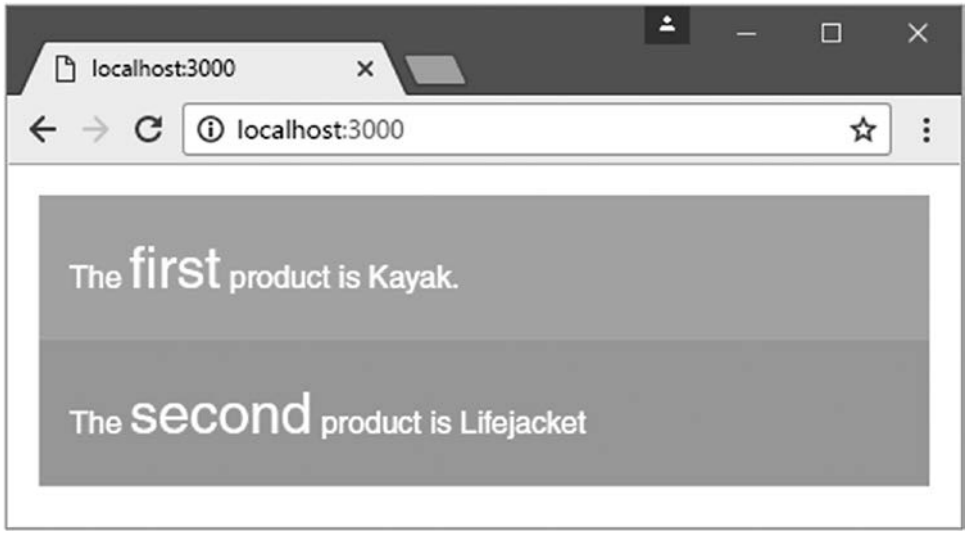


Рис. 12.10. Назначение отдельных стилевых свойств

Назначение стилей директивой ngStyle

Директива `ngStyle` позволяет задать несколько стилевых свойств при помощи объекта-карты (по аналогии с тем, как работает директива `ngClass`). В листинге 12.16 добавляется метод компонента, который возвращает карту со стилевыми настройками.

Листинг 12.16. Создание объекта карты в файле component.ts

```
import { Component } from "@angular/core";
import { Model } from "./repository.model";

@Component({
    selector: "app",
    templateUrl: "app/template.html"
})
export class ProductComponent {
    model: Model = new Model();

    getClasses(key: number): string {
        let product = this.model.getProduct(key);
        return "p-a-1 " + (product.price < 50 ? "bg-info" : "bg-warning");
    }

    getStyles(key: number) {
        let product = this.model.getProduct(key);
        return {
            fontSize: "30px",
            "margin.px": 100,
            color: product.price > 50 ? "red" : "green"
        };
    }
}
```

Объект-карта, возвращаемый методом `getStyle`, показывает, что директива `ngStyle` может поддерживать оба формата, которые могут использоваться с привязками свойств — как с включением единиц в значение, так и с именем свойства. Вот как выглядит объект-карта, генерируемый методом `getStyles` для аргумента `key`, равного 1:

```
{
  "fontSize":"30px",
  "margin.px":100,
  "color":"red"
}
```

В листинге 12.17 показаны привязки данных в шаблоне, использующие директиву `ngStyle` с выражениями, вызывающими метод `getStyles`.

Листинг 12.17. Использование директивы ngStyle в файле template.html

```
<div class="p-a-1 bg-warning">
    The <span [ngStyle]="getStyles(1)">first</span>
     product is {{model.getProduct(1).name}}.
</div>
<div class="p-a-1 bg-info">
    The <span [ngStyle]="getStyles(2)">second</span>
     product is {{model.getProduct(2).name}}
</div>
```

В результате каждый элемент `span` получает адаптированный набор стилей в зависимости от аргумента, переданного методом `getStyles` (рис. 12.11).

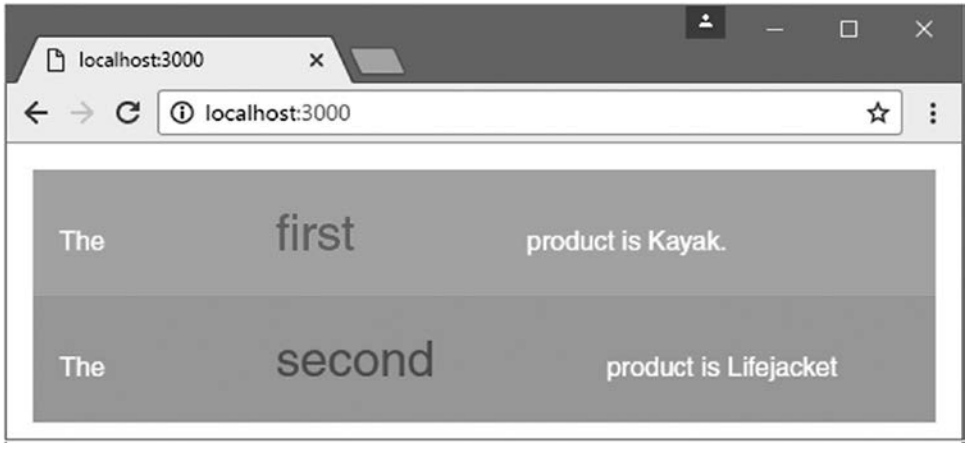


Рис. 12.11. Использование директивы ngStyle

Обновление данных в приложении

Когда вы только начинаете осваивать Angular, вам кажется, что с привязками данных слишком сложно работать: вы должны помнить, какая привязка нужна в той или иной ситуации. Стоят ли они затраченных усилий?

Привязки безусловно стоят вашего внимания. В основном это объясняется тем, что выражения привязок вычисляются заново при изменении данных, от которых они зависят. Например, если вы используете привязку со строковой интерполяцией для вывода значения свойства, привязка автоматически обновится при изменении значения свойства.

Для наглядности я немного забегу вперед и покажу, как организовать ручное управление процессом обновления. Этот прием не применяется в обычной разработке приложений Angular, но он убедительно демонстрирует важность привязок. В листинге 12.18 показаны изменения в компоненте, которые делают возможной эту демонстрацию.

Листинг 12.18. Настройка компонента в файле component.ts

```
import { ApplicationRef, Component } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";

@Component({
    selector: "app",
    templateUrl: "app/template.html"
})
export class ProductComponent {
    model: Model = new Model();

    constructor(ref: ApplicationRef) {
        (<any>window).appRef = ref;
        (<any>window).model = this.model;
```

```
    }

    getProductByPosition(position: number): Product {
        return this.model.getProducts()[position];
    }

    getClassesByPosition(position: number): string {
        let product = this.getProductByPosition(position);
        return "p-a-1 " + (product.price < 50 ? "bg-info" : "bg-warning");
    }
}
```

Здесь импортируется тип `ApplicationRef` из модуля `@angular/core`. В процессе выполнения начальной загрузки Angular создает объект `ApplicationRef`, представляющий приложение. Листинг 12.18 добавляет к компоненту конструктор, который получает объект `ApplicationRef` в аргументе, используя механизм внедрения зависимостей Angular, описанный в главе 19. Не углубляясь в подробности, скажу, что такое объявление аргумента конструктора сообщает Angular, что компонент должен получить объект `ApplicationRef` при создании нового экземпляра.

В этом конструкторе присутствуют две команды, которые делают возможной демонстрацию; с другой стороны, при использовании в реальном проекте вы бы лишились многих преимуществ от использования TypeScript и Angular.

```
...
(<any>window).appRef = ref;
(<any>window).model = this.model;
...
```

Эти команды определяют переменные в глобальном пространстве имен и присваивают им объекты `ApplicationRef` и `Model`. Загромождать глобальное пространство имен не рекомендуется, но доступ к объектам позволит управлять ими с консоли JavaScript в браузере, что важно в данном примере.

Другие методы, добавленные в конструктор, позволяют загрузить объект `Product` из репозитория по позиции (а не по ключу) и сгенерировать карту, зависящую от значения свойства `price`.

В листинге 12.19 показаны соответствующие изменения в шаблоне, использующие директиву `ngClass` для назначения принадлежности к классам, и привязку со строковой интерполяцией для вывода значения свойства `Product.name`.

Листинг 12.19. Подготовка к изменениям в файле template.html

```
<div [ngClass]="getClassesByPosition(0)">
    The first product is {{getProductByPosition(0).name}}.
</div>
<div [ngClass]="getClassesByPosition(1)">
    The second product is {{getProductByPosition(1).name}}
</div>
```

Сохраните изменения в компоненте и шаблоне. Когда браузер перезагрузит страницу, введите следующую команду в консоли JavaScript браузера и нажмите Return:

```
model.products.shift()
```

Команда вызывает для массива объектов `Product` модели метод `shift`, который удаляет из массива первый элемент и возвращает результат. Никаких изменений вы пока не увидите, потому что Angular еще не знает о том, что модель изменилась. Чтобы приказать Angular проверить изменения, введите следующую команду в консоли JavaScript в браузере и нажмите Return:

```
appRef.tick()
```

Метод `tick` запускает процесс обнаружения изменений: Angular просматривает данные в приложении и выражения в привязках данных и обрабатывает любые изменения. Привязки данных в шаблоне используют конкретные индексы массива для отображения данных, и после исключения объекта из модели привязки обновляются для вывода новых значений (рис. 12.12).

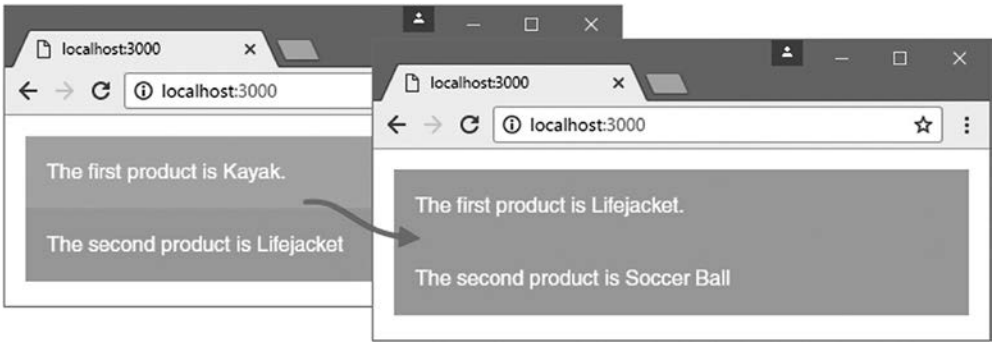


Рис. 12.12. Ручное обновление модели приложения

Задумайтесь над тем, что происходит при запуске процесса обнаружения изменений. Angular повторно обрабатывает выражения в привязках шаблона и обновляет их значения. В свою очередь, директива `ngClass` и привязка со строковой интерполяцией изменяют конфигурацию своих управляющих элементов с изменением состава назначенных им классов и отображением нового контента.

Это происходит из-за того, что привязки данных Angular являются «живыми»: связь между выражением, целью и управляющим элементом продолжает существовать после того, как исходный контент был выведен для пользователя, и динамически отражает изменения в состоянии приложения. Откровенно говоря, этот эффект производит намного большее впечатление, когда вам не приходится вносить изменения с консоли JavaScript. В главе 14 я объясню, как в Angular пользователь может инициировать изменения при помощи событий и форм.

Итоги

В этой главе я описал структуру привязок данных Angular и показал, как они используются для создания связей между данными приложения и элементами HTML, которые отображаются для пользователя. Вы узнали, как используются две встроенные директивы: `ngClass` и `ngStyle`. В следующей главе вы узнаете, как работают остальные встроенные директивы.

13 Встроенные директивы

В этой главе рассматриваются встроенные директивы, отвечающие за некоторые часто используемые возможности, применяемые при создании веб-приложений: избирательное включение контента, выбор между фрагментами контента и повторение контента. Также будут описаны некоторые ограничения, накладываемые Angular для выражений в односторонних привязках и реализующие их директивы. В табл. 13.1 встроенные директивы шаблонов представлены в контексте.

Таблица 13.1. Встроенные директивы

Вопрос	Ответ
Что это такое?	Встроенные директивы, описанные в этой главе, предназначены для избирательного включения контента, выбора между фрагментами контента и повторения контента для каждого элемента в массиве. Также существуют директивы для назначения стилей элемента и определения принадлежности к классам (см. главу 13)
Для чего нужны встроенные директивы?	Директивы обеспечивают выполнение самых распространенных и фундаментальных задач разработки веб-приложений и закладывают основу для адаптации контента, отображаемого для пользователя, на основании данных в приложении
Как они используются?	Директивы применяются к элементам HTML в шаблонах. Многочисленные примеры встречаются в этой главе (и в оставшейся части книги)
Есть ли у них недостатки или скрытые проблемы?	Вы должны помнить, что одни из этих директив (включая ngIf и ngFor) должны снабжаться префиксом *, тогда как другие (включая ngClass, ngStyle и ngSwitch) заключаются в квадратные скобки. Причины объясняются во врезке «Директивы микрошаблонов», но об этих особенностях синтаксиса легко забыть; в этом случае вы рискуете получить неожиданные результаты
Есть ли альтернативы?	Разработчик может создавать собственные директивы (этот процесс будет описан в главах 15 и 16), но встроенные директивы работают эффективно и тщательно протестированы. В большинстве приложений рекомендуется использовать встроенные директивы (кроме тех случаев, когда они не могут предоставить необходимую функциональность)

В табл. 13.2 приведена краткая сводка материала главы.

Таблица 13.2. Сводка материала главы

Проблема	Решение	Листинг
Условное отображение контента в зависимости от выражения привязки данных	Используйте директиву ngIf	1–3
Выбор между несколькими вариантами контента в зависимости от значения выражения привязки данных	Используйте директиву ngSwitch	4, 5
Генерирование блока контента для каждого объекта, определяемого выражением привязки данных	Используйте директиву ngFor	6–12
Повторение блока контента	Используйте директиву ngTemplateOutlet	13–14
Предотвращение ошибок при использовании шаблонов	Избегайте побочных эффектов выражений привязки данных, влияющих на состояние приложения	15–19
Предотвращение контекстных ошибок	Следите за тем, чтобы выражения привязки данных изменяли только свойства и методы, предоставляемые компонентом шаблона	20–22

Подготовка проекта

В этой главе мы продолжим использовать проект example, созданный в главе 11 и измененный в главе 12. Чтобы подготовиться к теме этой главы, я внес изменения в класс компонента (листинг 13.1), удалил ненужные фрагменты и добавил новые методы и свойство.

Листинг 13.1. Изменения в файле component.ts

```
import { ApplicationRef, Component } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";

@Component({
    selector: "app",
    templateUrl: "app/template.html"
})
export class ProductComponent {
    model: Model = new Model();

    constructor(ref: ApplicationRef) {
        (<any>window).appRef = ref;
        (<any>window).model = this.model;
```

```
    }

    getProductByPosition(position: number): Product {
        return this.model.getProducts()[position];
    }

    getProduct(key: number): Product {
        return this.model.getProduct(key);
    }

    getProducts(): Product[] {
        return this.model.getProducts();
    }

    getProductCount(): number {
        return this.getProducts().length;
    }

    targetName: string = "Kayak";
}
```

В листинге 13.2 показано содержимое файла шаблона для вывода количества товаров в модели данных с вызовом нового метода `getProductCount` компонента.

Листинг 13.2. Содержимое файла template.html

```
<div class="bg-info p-a-1">
    There are {{getProductCount()}} products.
</div>
```

Выполните следующую команду из папки example, чтобы запустить компилятор TypeScript и сервер HTTP для разработки:

```
npm start
```

Открывается новое окно браузера, в котором загружается файл index.html из папки example (рис. 13.1).

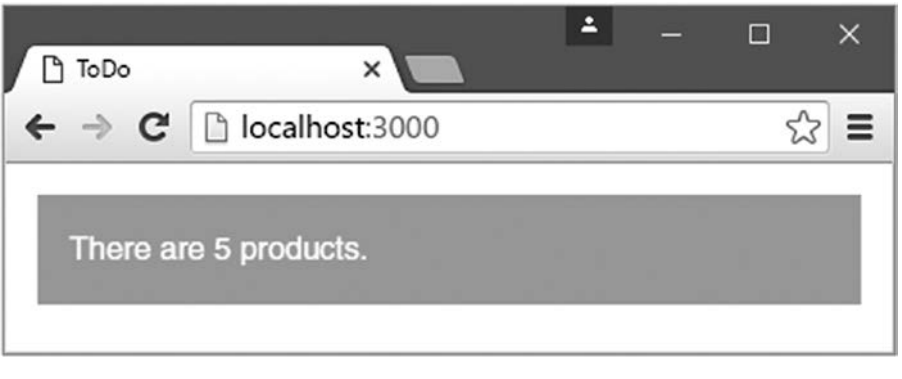


Рис. 13.1. Запуск приложения

Использование встроенных директив

Angular включает набор встроенных директив для реализации функциональности, часто используемой в веб-приложениях, чтобы разработчикам не приходилось заново реализовывать ее в своих проектах. В табл. 13.3 описаны доступные директивы, использование которых будет продемонстрировано далее (кроме директив `ngClass` и `ngStyle`, рассмотренных в главе 12).

Таблица 13.3. Встроенные директивы

Пример	Описание
<div *ngIf="expr">	Директива ngIf включает элемент и его контент в документ HTML в том случае, если результат выражения равен true. Звездочка перед именем означает, что директива является директивой микрошаблона (см. врезку «Директивы микрошаблонов»)
<div [ngSwitch]="expr"> <span *ngSwitchCase="expr"> <span *ngSwitchDefault>	Директива ngSwitch используется для выбора элементов, включаемых в документ HTML, на основании результата выражения; результат сравнивается с результатами отдельных выражений, определяемых в директивах ngSwitchCase. Если ни одно из значений ngSwitchCase не совпадает, используется элемент, к которому применена директива ngSwitchDefault. Звездочки перед именами ngSwitchCase и ngSwitchDefault означают, что директивы являются директивами микрошаблонов (см. врезку «Директивы микрошаблонов»)
<div *ngFor="#item of expr">	Директива ngFor генерирует повторяющийся набор элементов для каждого объекта в массиве. Звездочка перед именем означает, что директива является директивой микрошаблона (см. врезку «Директивы микрошаблонов»)
<template [ngTemplateOutlet]= "myTempl"> </ template> 	Директива ngTemplateOutlet используется для повторения блока контента в шаблоне. Директива ngClass используется для управления принадлежностью к классам (см. главу 12). Директива ngStyle используется для управления стилями, применяемыми непосредственно к элементам, — в отличие от применения стилей через классы (см. главу 12)

Директива ngIf

Директива `ngIf` — простейшая из встроенных директив, которая включает фрагмент HTML в документ, если результат вычисления выражения равен `true` (листинг 13.3).

Листинг 13.3. Использование директивы ngIf в файле template.html

```
<div class="bg-info p-a-1">
    There are {{getProductCount()}} products.
</div>

<div *ngIf="getProductCount() > 4" class="bg-info p-a-1 m-t-1">
    There are more than 4 products in the model
</div>
<div *ngIf="getProductByPosition(0).name != 'Kayak'" class="bg-info p-a-1 m-t-1">
    The first product isn't a Kayak
</div>
```

Директива `ngIf` применяется к двум элементам `div`; выражения проверяют количество объектов `Product` в модели, а также то, содержит ли свойство `name` первого объекта `Product` строку `Kayak`.

Результат первого выражения равен `true`; это означает, что элемент `div` и его контент будут включены в документ HTML. Результат второго выражения равен `false`; это означает, что второй элемент `div` включаться не будет. На рис. 13.2 показан результат.

ПРИМЕЧАНИЕ

Директива ngIf добавляет и удаляет элементы из документа HTML, вместо того чтобы просто отображать и скрывать их. Если вы хотите оставить элементы на своих местах, управляя их видимостью, используйте привязки свойств или стилей, описанные в главе 12: либо заданием свойства элемента hidden, либо назначением стилевому свойству display значения none.

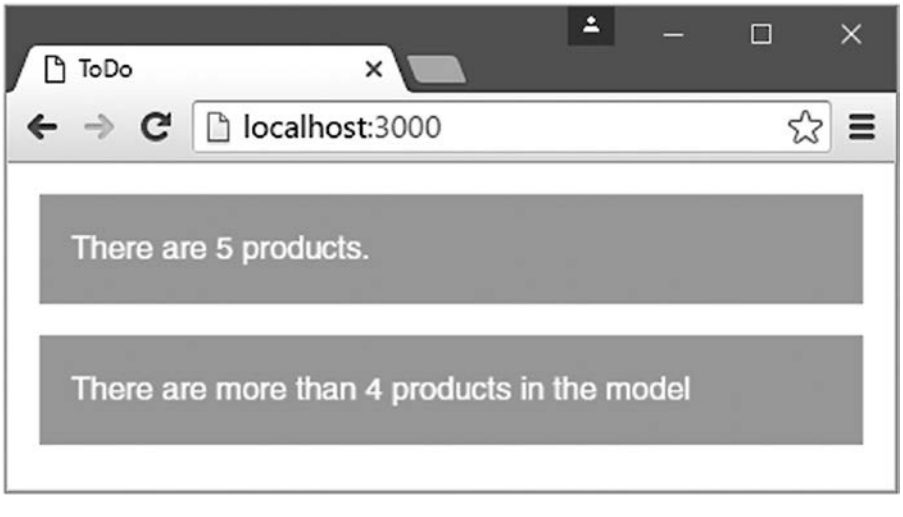


Рис. 13.2. Использование директивы ngIf

ДИРЕКТИВЫ МИКРОШАБЛОНОВ

Некоторые директивы (такие, как ngFor, ngIf и вложенные директивы, используемые с ngSwitch) записываются с префиксом *: например, *ngFor, *ngIf и *ngSwitch. Звездочка является сокращенным обозначением для директив, полагающихся на контент, предоставляемый как часть шаблона, — так называемый микрошаблон. Директивы, использующие микрошаблоны, называются структурными директивами; мы вернемся к этому описанию в главе 16, когда вы научитесь их создавать.

В листинге 13.3 директива ngIf применяется к элементам div; директива будет использовать элемент div и его контент как микрошаблон для каждого из обрабатываемых объектов. Во внутренней реализации Angular расширяет микрошаблон и директиву следующим образом:

```
...
<template ngIf="model.getProductCount() > 4">
    <div class="bg-info p-a-1 m-t-1">
        There are more than 4 products in the model
    </div>
</template>
...
```

Вы можете использовать в своих шаблонах любой вариант, но при использовании компактного синтаксиса следует помнить о необходимости включения звездочки. Я объясню, как создавать собственные директивы микрошаблонов, в главе 14.

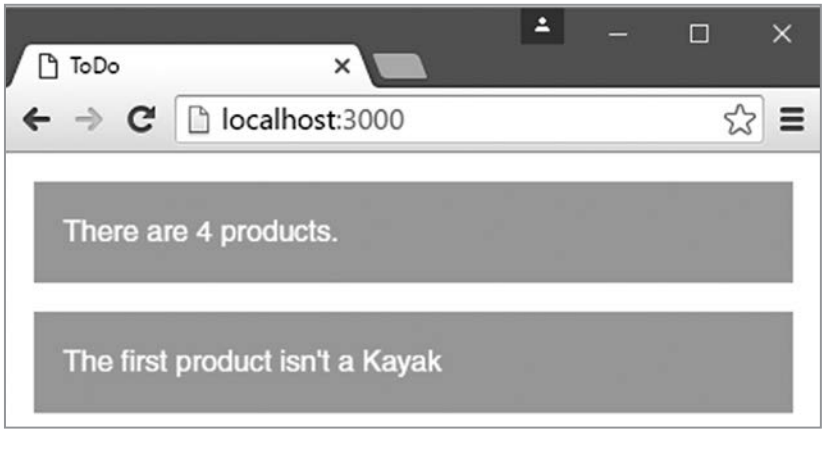


Рис. 13.3. Эффект повторного вычисления выражений в директивах

Как и все директивы, выражение в `ngIf` заново обрабатывается в соответствии с изменениями в модели данных. Выполните следующие команды в консоли JavaScript браузера, чтобы удалить первый объект данных и запустить процесс обнаружения изменений:

```
model.products.shift()
appRef.tick()
```

Изменение модели заключается в удалении первого элемента div, потому что теперь объектов Product становится слишком мало, и добавлении второго элемента div, потому что свойство name первого объекта Product в массиве отлично от Kayak. Изменения показаны на рис. 13.3.

Директива ngSwitch

Директива ngSwitch выбирает один из нескольких элементов в зависимости от результата выражения по аналогии с командой JavaScript switch. В листинге 13.4 продемонстрировано применение директивы ngSwitch для выбора элемента в зависимости от количества объектов в модели.

Листинг 13.4. Использование директивы ngSwitch в файле template.html

```
<div class="bg-info p-a-1">
    There are {{getProductCount()}} products.
</div>

<div class="bg-info p-a-1 m-t-1" [ngSwitch]="getProductCount()">
    <span *ngSwitchCase="2">There are two products</span>
    <span *ngSwitchCase="5">There are five products</span>
    <span *ngSwitchDefault>This is the default</span>
</div>
```

Синтаксис директивы ngSwitch на первый взгляд кажется довольно запутанным. Элемент, к которому применяется директива ngSwitch, всегда включается в документ HTML, а имя директивы не снабжается префиксом *. Оно должно заключаться в квадратные скобки:

```
...
<div class="bg-info p-a-1 m-t-1" [ngSwitch]="getProductCount()">
...
```

Каждый из внутренних элементов (в данном случае это элементы span) является микрошаблоном, а директивы с указанием результата целевого выражения снабжаются префиксом *:

```
...
<span *ngSwitchCase="5">There are five products</span>
...
```

Директива ngSwitchCase задает результат конкретного выражения. Если результат выражения ngSwitch совпадает с указанным результатом, то элемент и его содержимое включаются в документ HTML. Если же результат вычисления выражения не совпадает с указанным, то элемент и его контент не будут включены в документ HTML.

Директива ngSwitchDefault применяется к элементу по умолчанию (аналог метки default в конструкции switch языка JavaScript), который включается в документHTML, если результат выражения не совпал ни с одним из результатов, указанных в директивах ngSwitchCase.

Для исходных данных приложения директивы из листинга 13.4 генерируют следующую разметку HTML:

```
<div class="bg-info p-a-1 m-t-1" ng-reflect-ng-switch="5">
    <span>There are five products</span>
</div>
```

Элемент `div`, к которому применяется директива `ngSwitch`, всегда включается в документ HTML. Для исходных данных модели также будет включен элемент `span`, у которого директива `ngSwitchCase` возвращает 5; в итоге в браузере будет выведен результат, изображенный слева на рис. 13.4.

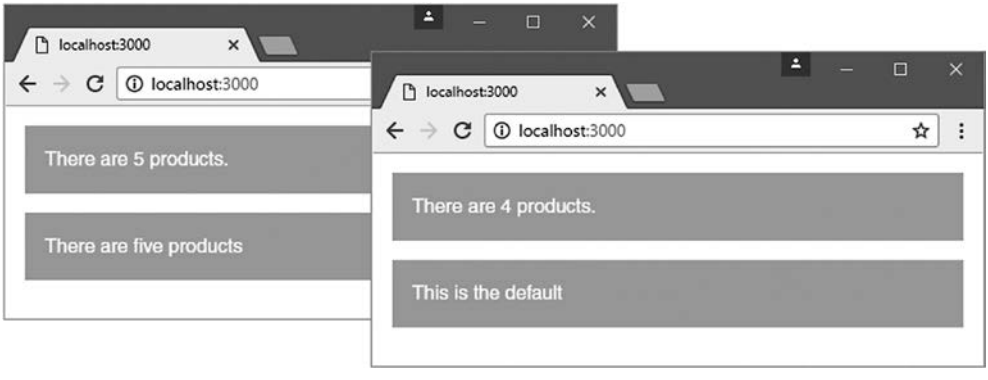


Рис. 13.4. Использование директивы ngSwitch

Привязка `ngSwitch` реагирует на изменения в модели данных; чтобы убедиться в этом, выполните следующие команды в консоли JavaScript браузера:

```
model.products.shift()
appRef.tick()
```

Эти команды удаляют первый объект из модели и заставляют Angular выполнить процесс обнаружения изменений. Ни один из результатов двух директив `ngSwitchCase` не совпадает с результатом выражения `getProductCount`, поэтому в документ HTML включается элемент `ngSwitchDefault` (см. рис. 13.4, справа).

Предотвращение проблем с литералами

Типичная проблема встречается при использовании директивы `ngSwitchCase` со строковыми литералами. Чтобы добиться правильного результата, необходимо действовать очень внимательно (листинг 13.5).

Листинг 13.5. Переключение по значениям компонентов и строковым литералам в файле template.html

```
<div class="bg-info p-a-1">
    There are {{getProductCount()}} products.
```

```
</div>

<div class="bg-info p-a-1 m-t-1" [ngSwitch]="getProduct(1).name">
    <span *ngSwitchCase="targetName">Kayak</span>
    <span *ngSwitchCase="'Lifejacket'">Lifejacket</span>
    <span *ngSwitchDefault>Other Product</span>
</div>
```

Значения, указанные в директивах `ngSwitchCase`, также являются выражениями; это означает, что в них можно вызывать методы, выполнять простые встроенные операции и читать значения свойств, как и в базовых привязках данных.

Например, следующее выражение приказывает Angular включить элемент `span`, к которому применяется директива, если результат вычисления выражения `ngSwitch` совпадает со значением свойства `targetName`, определяемого компонентом:

```
...
<span *ngSwitchCase="targetName">Kayak</span>
...
```

Если вы хотите сравнить результат с конкретной строкой, заключите ее в кавычки и апострофы:

```
...
<span *ngSwitchCase="'Lifejacket'">Lifejacket</span>
...
```

Это выражение приказывает Angular включить элемент `span`, у которого значение выражения `ngSwitch` равно строковому литералу `Lifejacket`. В итоге будет получен результат, показанный на рис. 13.5.

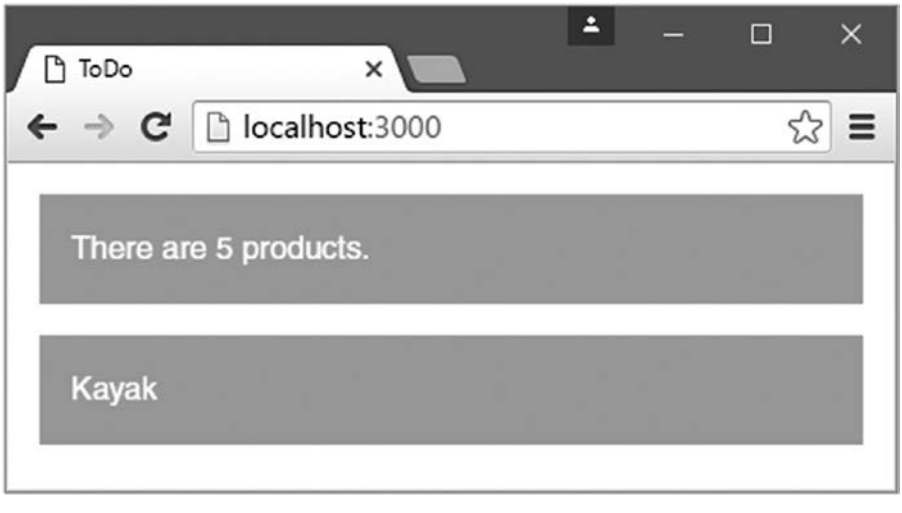


Рис. 13.5. Использование выражений и строковых литералов с директивой ngSwitch

Директива ngFor

Директива ngFor повторяет блок контента для каждого объекта в последовательности объектов; таким образом реализуется шаблонный аналог цикла foreach и предоставляется механизм перебора коллекции объектов. В листинге 13.6 директива ngFor используется для заполнения таблицы с генерированием строки для каждого объекта Product в модели.

Листинг 13.6. Использование директивы ngFor в файле template.html

```
<div class="bg-info p-a-1">
    There are {{getProductCount()}} products.
</div>

<table class="table table-sm table-bordered m-t-1">
    <tr><th>Name</th><th>Category</th><th>Price</th></tr>
    <tr *ngFor="let item of getProducts()">
        <td>{{item.name}}</td>
        <td>{{item.category}}</td>
        <td>{{item.price}}</td>
    </tr>
</table>
```

Выражение в директиве ngFor сложнее, чем в других встроенных директивах, но и оно становится более понятным, когда вы поймете, как разные части взаимодействуют друг с другом. Директива, которая используется в примере, выглядит так:

```
...
<tr *ngFor="let item of getProducts()">
...
```

Звездочка перед именем обязательна, потому что директива использует микрошаблон (см. врезку «Директивы микрошаблонов»). Происходящее станет более понятным по мере изучения Angular, а пока просто запомните, что для использования директивы необходима звездочка (или, как это часто бывает со мной, забудьте об этом, пока не увидите ошибку на консоли JavaScript в браузере — и *тогда* вспомните).

Что касается самого выражения, оно состоит из двух частей, объединенных ключевым словом of. Правая часть выражения определяет источник данных, объекты которого участвуют в переборе.

```
...
<tr *ngFor="let item of getProducts()">
...
```

В этом примере в качестве источника данных указывается метод getProducts компонента, который позволяет сгенерировать контент для каждого объекта Product в модели. То, что находится в правой части, является самостоятельным выражением; таким образом в нем можно подготовить данные или выполнить простые операции с шаблоном.

В левой части выражения `ngFor` определяется *переменная шаблона*, на что указывает ключевое слово `let`; этот механизм используется для передачи данных между элементами в шаблонах Angular.

```
...
<tr *ngFor="let item of getProducts()">
...
```

Директива `ngFor` присваивает переменной каждый объект в источнике данных. Таким образом, переменная доступна для использования вложенными элементами. Локальная переменная шаблона в этом примере называется `item`, и она используется для обращения к свойствам объекта `Product` в элементах `td`:

```
...
<td>{{item.name}}</td>
...
```

Объединим все сказанное: директива в этом примере приказывает Angular перебрать объекты, возвращаемые методом `getProducts` компонента, присваивает каждый из них переменной с именем `item`, после чего генерирует элемент `tr` и его дочерние элементы `td`, вычисляя содержащиеся в них шаблонные выражения.

В примере из листинга 13.6 результатом является таблица, в которой директива `ngFor` генерирует строку для каждого объекта `Product` в модели; каждая строка таблицы отображает элементы `td` для вывода свойств `name`, `category` и `price`, как показано на рис. 13.6.

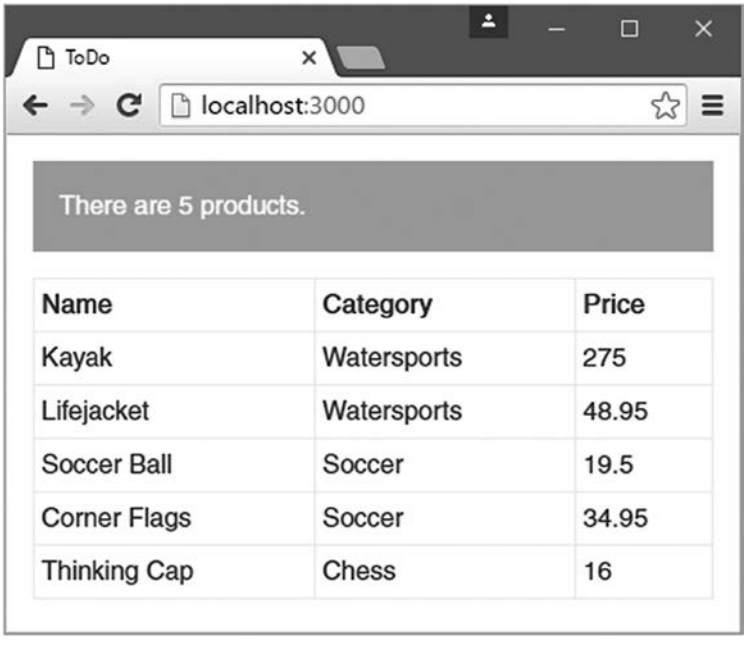


Рис. 13.6. Использование директивы ngFor для создания строк таблицы

Другие переменные шаблонов

Самая важная переменная шаблона обозначает обрабатываемый объект данных (`item` в предыдущем примере). Директива `ngFor` поддерживает ряд других значений, которые могут присваиваться переменным для последующего обращения во вложенных элементах HTML. Они описаны в табл. 13.4, а их использование продемонстрировано в последующих разделах.

Таблица 13.4. Локальные значения шаблонов ngFor

Имя	Описание
index	Значение типа number; содержит позицию текущего объекта
odd	Значение типа boolean; возвращает true, если текущий объект находится в нечетной позиции источника данных
even	Значение типа boolean; возвращает true, если текущий объект находится в четной позиции источника данных
first	Значение типа boolean; возвращает true, если текущий объект находится в первой позиции источника данных
last	Значение типа boolean; возвращает true, если текущий объект находится в последней позиции источника данных

Использование значения index

Значению `index` присваивается позиция текущего объекта данных, увеличиваемая для каждого объекта в источнике данных. В листинге 13.7 определяется таблица, которая заполняется директивой `ngFor`; значение `index` присваивается локальной переменной шаблона с именем `i`, которая затем используется в привязке со строковой интерполяцией.

Листинг 13.7. Использование значения index в файле template.html

```
<div class="bg-info p-a-1">
    There are {{getProductCount()}} products.
</div>

<table class="table table-sm table-bordered m-t-1">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
    <tr *ngFor="let item of getProducts(); let i = index">
        <td>{{i +1}}</td>
        <td>{{item.name}}</td>
        <td>{{item.category}}</td>
        <td>{{item.price}}</td>
    </tr>
</table>
```

В выражение `ngFor` добавляется новое подвыражение, отделенное от существующего символом «точка с запятой» (`;`). В новом подвыражении ключевое слово `let` используется для присваивания значения `index` локальной переменной с именем `i`:

```
...
<tr *ngFor="let item of getProducts(); let i = index">
...
```

Это позволяет обратиться к значению во вложенных элементах с использованием привязок:

```
...
<td>{{i + 1}}</td>
...
```

Значение `index` начинается с нуля, а увеличение значения на 1 создает простой счетчик; результат показан на рис. 13.7.

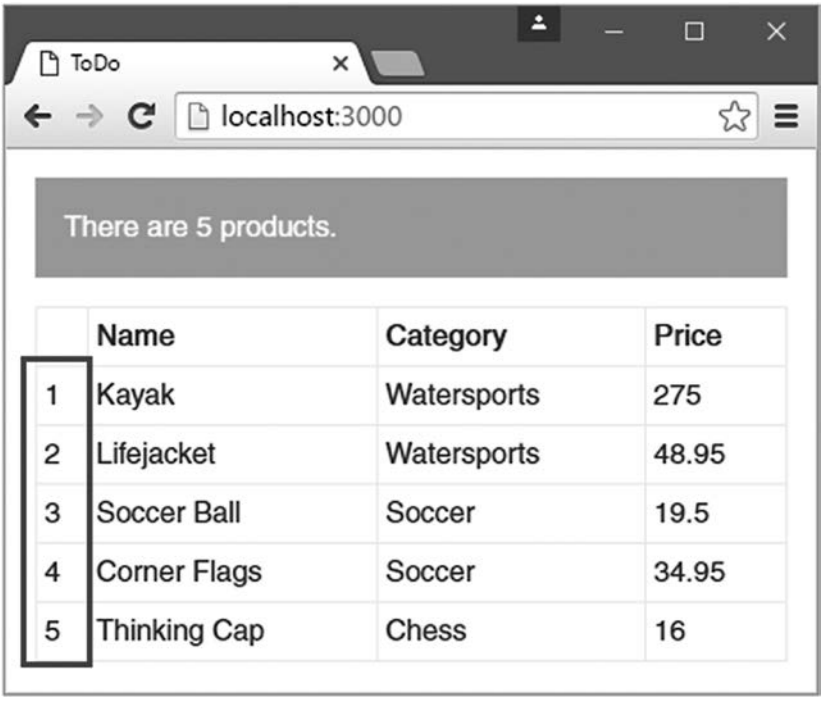


Рис. 13.7. Использование значения index

Использование значений odd и even

Значение `odd` истинно, если значение `index` для элемента данных нечетно. И наоборот, значение `even` истинно, если значение `index` для элемента данных четно. Как правило, в приложении достаточно использовать либо `odd`, либо `even`; это логические значения, и значение `odd` истинно, если значение `even` ложно, и наоборот. В листинге 13.8 значение `odd` используется для управления классами элементов `tr` таблицы.

Листинг 13.8. Использование значения odd в файле template.html

```
<div class="bg-info p-a-1">
    There are {{getProductCount()}} products.
</div>

<table class="table table-sm table-bordered m-t-1">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
    <tr *ngFor="let item of getProducts(); let i = index; let odd = odd"
            [class.bg-primary]="odd" [class.bg-info]="!odd">
        <td>{{i + 1}}</td>
        <td>{{item.name}}</td>
        <td>{{item.category}}</td>
        <td>{{item.price}}</td>
    </tr>
</table>
```

Через точку с запятой (`;`) в выражение `ngFor` было добавлено подвыражение, присваивающее значение `odd` локальной переменной шаблона, которая также называется `odd`.

```
...
<tr *ngFor="let item of getProducts(); let i = index; let odd = odd"
    [class.bg-primary]="odd" [class.bg-info]="!odd">
...
```

На первый взгляд присваивание кажется избыточным, но к значениям `ngFor` нельзя обращаться напрямую; необходимо использовать локальную переменную, пусть одноименную. Я использую привязку класса для назначения чередующимся строкам таблицы классов `bg-primary` и `bg-info` — классов цвета фона Bootstrap, которые окрашивают строки таблицы с чередованием цветов (рис. 13.8).

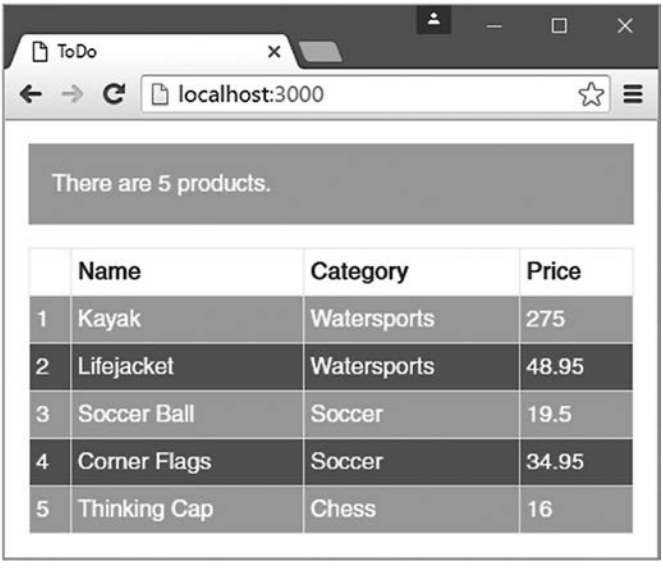


Рис. 13.8. Использование значения odd

РАСШИРЕНИЕ ДИРЕКТИВЫ *NGFOR

Обратите внимание: в листинге 13.8 переменная шаблона может использоваться в выражениях, применяемых к тому же элементу tr, который ее определяет. Это возможно благодаря тому, что ngFor является директивой микрошаблона (на что указывает префикс * перед именем), поэтому Angular расширяет HTML до следующего вида:

```
...
<table class="table table-sm table-bordered m-t-1">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
    <template ngFor let-item [ngForOf]="getProducts()"
            let-i="index" let-odd="odd">
        <tr [class.bg-primary]="odd" [class.bg-info]="!odd">
            <td>{{i + 1}}</td>
            <td>{{item.name}}</td>
            <td>{{item.category}}</td>
            <td>{{item.price}}</td>
        </tr>
    </template>
</table>
...
```

Как видите, элемент template определяет переменные с использованием несколько неуклюжих атрибутов let-<имя>, к которым затем могут обращаться внутренние элементы tr и td. Как это часто бывает в Angular, то, что на первый взгляд кажется каким-то волшебством, становится предельно ясным, когда вы начинаете понимать, что происходит «за кулисами»; эти возможности подробно рассматриваются в главе 16. Убедительный довод в пользу синтаксиса *ngFor заключается в том, что он предоставляет более элегантный способ записи выражения директивы, особенно при наличии нескольких переменных шаблонов.

Использование значений first и last

Значение `first` истинно только для первого объекта последовательности, предоставляемой источником данных, и ложно для всех остальных объектов. И наоборот, значение `last` истинно только для последнего объекта последовательности. В листинге 13.9 эти значения используются для особой обработки первого и последнего объекта последовательности.

Листинг 13.9. Использование значений first и last в файле template.html

```
<div class="bg-info p-a-1">
    There are {{getProductCount()}} products.
</div>

<table class="table table-sm table-bordered m-t-1">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
    <tr *ngFor="let item of getProducts(); let i = index; let odd = odd;
            let first = first; let last = last"
            [class.bg-primary]="odd" [class.bg-info]="!odd"
            [class.bg-warning]="first || last">
```

```
        <td>{{i + 1}}</td>
        <td>{{item.name}}</td>
        <td>{{item.category}}</td>
        <td *ngIf="!last">{{item.price}}</td>
    </tr>
</table>
```

Новые подвыражения в выражении `ngFor` присваивают значения `first` и `last` переменным шаблона с именем `first` и `last`. Затем эти переменные используются привязкой класса элемента `tr`, которая назначает элементу класс `bg-warning`, если одно из двух значений истинно. Директива `ngIf` одного из элементов `td` исключает элемент последнего объекта источника данных. Результат показан на рис. 13.9.

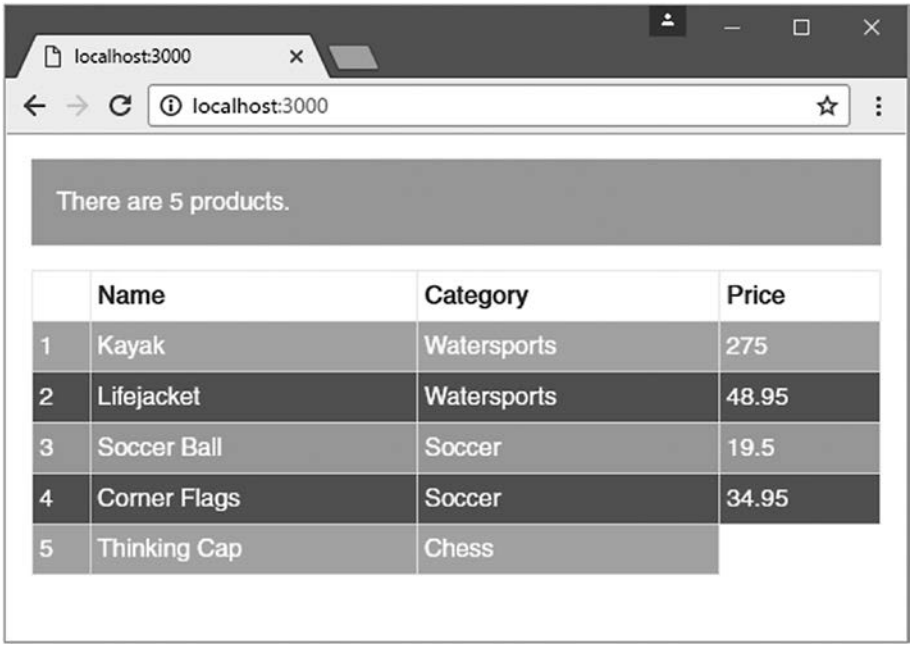


Рис. 13.9. Использование значений first и last

Минимизация операций с элементами

При изменении модели данных директива `ngFor` вычисляет свое выражение и обновляет элемент, представляющий объекты данных. Процесс обновления может быть весьма затратным, особенно если источник данных заменяется другим — с другими объектами, представляющими те же данные. Может показаться, что замена источника данных — довольно странная затея, но в веб-приложениях такое происходит довольно часто, особенно при получении данных от веб-служб (вроде той, которая будет описана в главе 24). Прежние значения данных представляются новыми объектами, и это создает проблемы с эффективностью приложений

Angular. Чтобы продемонстрировать суть проблемы, я добавлю в компонент метод, заменяющий один из объектов `Product` в модели данных (листинг 13.10).

Листинг 13.10. Замена объекта в файле repository.model.ts

```
import { Product } from "./product.model";
import { SimpleDataSource } from "./datasource.model";

export class Model {
    private dataSource: SimpleDataSource;
    private products: Product[];
    private locator = (p:Product, id:number) => p.id == id;

    constructor() {
        this.dataSource = new SimpleDataSource();
        this.products = new Array<Product>();
        this.dataSource.getData().forEach(p => this.products.push(p));
    }

    // ...Другие методы опущены для краткости...

    swapProduct() {
        let p = this.products.shift();
        this.products.push(new Product(p.id, p.name, p.category, p.price));
    }
}
```

Метод `swapProduct` исключает первый объект из массива и добавляет новый объект с теми же значениями свойств `id`, `name`, `category` и `price` (пример представления значений данных новым объектом).

Выполните следующие команды в консоли JavaScript браузера, чтобы изменить модель данных и запустить процесс обнаружения изменений:

```
model.swapProduct()
appRef.tick()
```

Когда директива `ngFor` анализирует свой источник данных, она видит, что для отражения изменений в данных следует выполнить две операции. Первая операция — уничтожение элементов HTML, представляющих первый объект в массиве. Вторая операция — создание нового набора элементов HTML для представления нового объекта в конце массива.

Angular не может определить, что объекты данных, о которых идет речь, сохранили прежние значения и задачу можно было бы решить более эффективно простой перестановкой существующих элементов в документе HTML.

В данном примере задействованы только два элемента, но проблема становится куда более серьезной при обновлении данных приложения из внешнего источника данных средствами Ajax, когда заменяться могут все объекты модели данных. Не зная о том, что количество реальных изменений может быть незначительным, директива `ngFor` вынуждена уничтожать все элементы HTML и создавать их заново. Это требует значительных затрат вычислительных ресурсов и времени.

Для повышения эффективности обновления можно определить метод компонента, который поможет Angular определить, когда два разных объекта представляют одни и те же данные (листинг 13.11).

Листинг 13.11. Добавление метода сравнения объектов в файле component.ts

```
import { ApplicationRef, Component } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";

@Component({
    selector: "app",
    templateUrl: "app/template.html"
})

export class ProductComponent {
    model: Model = new Model();

    // ...Конструктор и методы опущены для краткости...

    getKey(index: number, product: Product) {
        return product.id;
    }
}
```

Метод должен определять два параметра: позицию объекта в источнике данных и объект данных. Результат метода однозначно идентифицирует объект; два объекта считаются равными, если они дают одинаковые результаты.

Два объекта `Product` будут считаться равными, если они имеют одинаковые значения `id`. Чтобы выражение `ngFor` использовало метод сравнения, добавьте в выражение условие `trackBy` (листинг 13.12).

Листинг 13.12. Определение метода проверки равенства в файле template.html

```
<div class="bg-info p-a-1">
    There are {{getProductCount()}} products.
</div>

<table class="table table-sm table-bordered m-t-1">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
    <tr *ngFor="let item of getProducts(); let i = index; let odd = odd;
            let first = first; let last = last; trackBy:getKey"
            [class.bg-primary]="odd" [class.bg-info]="!odd"
            [class.bg-warning]="first || last">
        <td>{{i + 1}}</td>
        <td>{{item.name}}</td>
        <td>{{item.category}}</td>
        <td *ngIf="!last">{{item.price}}</td>
    </tr>
</table>
```

С такими изменениями директива `ngFor` будет знать, что объект `Product`, удаленный из массива методом `swapProduct` из листинга 13.12, эквивалентен объекту, добавленному в массив, хотя данные представлены разными объектами. Вместо того чтобы тратить ресурсы на удаление и создание, Angular может переместить существующие элементы; это намного более простая и быстрая операция.

При этом в элементы могут вноситься изменения (например, директивой `ngIf`, которая удаляет один из элементов `td`, потому что новый объект оказывается последним в источнике данных), но даже эти операции выполняются быстрее, чем если бы объекты рассматривались как совершенно самостоятельные.

ТЕСТИРОВАНИЕ МЕТОДА ПРОВЕРКИ РАВЕНСТВА

Проверить, приводит ли метод проверки равенства к желаемому эффекту, непросто. Лучший способ, который мне удалось обнаружить, основан на использовании инструментария разработчика F12 — в данном случае в браузере Chrome.

После того как приложение будет загружено, щелкните правой кнопкой мыши на элементе td со словом Kayak в окне браузера и выберите в контекстном меню команду Inspect. На экране появляется окно инструментария разработчика и панель Elements.

Щелкните на кнопке с многоточием (...) в левой части, выберите в меню команду Add Attribute. Добавьте атрибут id со значением old. В результате элемент принимает следующий вид:

```
<td id="old">Kayak</td>
```

Добавление атрибута id позволит обращаться к объекту, представляющему элемент HTML, с консоли JavaScript. Переключитесь на панель консоли и введите следующую команду:

```
window.old
```

Нажмите клавишу Return. Браузер находит элемент по значению атрибута id и выводит следующий результат:

```
<td id="old">Kayak</td>
```

Теперь выполните в консоли Javascript следующие команды (нажмите Return после каждой команды):

```
model.swapProduct()
appRef.tick()
```

После обработки изменений модели данных выполните следующую команду в консоли JavaScript, чтобы определить, был ли перемещен или уничтожен элемент td с добавленным атрибутом id:

```
window.old
```

Если элемент был перемещен, он выводится на консоли следующим образом:

```
<td id="old">Kayak</td>
```

Если элемент был уничтожен, то элемента с атрибутом id не существует и браузер выведет слово undefined.

Использование директивы ngTemplateOutlet

Директива `ngTemplateOutlet` используется для повторения блока контента в заданной позиции, что может быть полезно для генерирования одного контента в нескольких местах с предотвращением дублирования. В листинге 13.13 продемонстрировано использование директивы.

Листинг 13.13. Использование директивы ngTemplateOutlet в файле template.html

```
<template #titleTemplate>
    <h4 class="p-a-1 bg-success">Repeated Content</h4>
</template>

<template [ngTemplateOutlet]="titleTemplate"></template>

<div class="bg-info p-a-1 m-a-1">
    There are {{getProductCount()}} products.
</div>

<template [ngTemplateOutlet]="titleTemplate"></template>
```

На первом шаге определяется шаблон с контентом, который должен повторяться с использованием директивы. Для этого используется элемент `template`, которому присваивается имя с использованием *ссылочной переменной*:

```
...
<template #titleTemplate let-title="title">
    <h4 class="p-a-1 bg-success">{{title}}</h4>
</template>
...
```

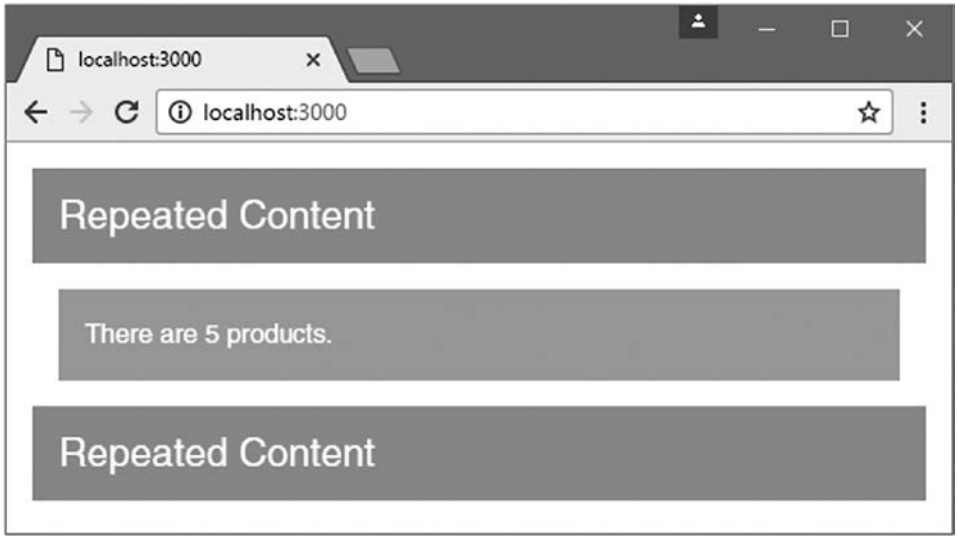


Рис. 13.10. Использование директивы ngTemplateOutlet

Обнаруживая ссылочную переменную, Angular присваивает ей элемент, в котором она была определена (в данном случае это элемент `template`).

На втором шаге контент вставляется в документ HTML при помощи директивы `ngTemplateOutlet`:

```
...
<template [ngTemplateOutlet]="titleTemplate"></template>
...
```

Выражением является имя ссылочной переменной, связанной со вставляемым контентом. Директива заменяет управляющий элемент содержимым заданного элемента `template`. Ни элемент `template`, содержащий повторяемый контент, ни управляющий элемент привязки не включается в документ HTML. На рис. 13.10 продемонстрировано применение директивы для повторения контента.

Предоставление контекстных данных

Директива `ngTemplateOutlet` может использоваться для передачи повторяющемуся контенту объекта контекста, который может использоваться с привязками данных, определенными в элементе `template` (листинг 13.14).

Листинг 13.14. Предоставление контекстных данных в файл template.html

```
<template #titleTemplate let-text="title">
    <h4 class="p-a-1 bg-success">{{text}}</h4>
</template>

<template [ngTemplateOutlet]="titleTemplate"
          [ngOutletContext]="{title: 'Header'}">
</template>

<div class="bg-info p-a-1 m-a-1">
    There are {{getProductCount()}} products.
</div>

<template [ngTemplateOutlet]="titleTemplate"
          [ngOutletContext]="{title: 'Footer'}">
</template>
```

Чтобы получить контекстные данные, элемент `template`, содержащий повторяющийся контент, определяет `let`-атрибут, задающий имя переменной (по аналогии с расширенным синтаксисом, использованным для директивы `ngFor`).

Результат выражения присваивает `let`-переменной значение:

```
...
<template #titleTemplate let-text="title">
...
```

`let`-атрибут в данном случае создает переменную с именем `text`, значение которой присваивается вычислением выражения `title`. Чтобы предоставить данные для вычисления выражения, элемент `template`, к которому была применена директива `ngTemplateOutlet`, предоставляет объект-карту:

```
...
<template [ngTemplateOutlet]="titleTemplate"
          [ngOutletContext]="{title: 'Footer'}">
</template>
...
```

Цель новой привязки, `ngOutletContext`, выглядит как еще одна директива, но на самом деле является примером *свойств ввода*, используемых некоторыми директивами для получения данных (эта тема более подробно рассматривается в главе 15). Выражение привязки представляет собой объект-карту, имя свойства которого соответствует `let`-атрибуту другого элемента `template`. Это позволяет адаптировать повторяющийся контент с использованием привязок (рис. 13.11).

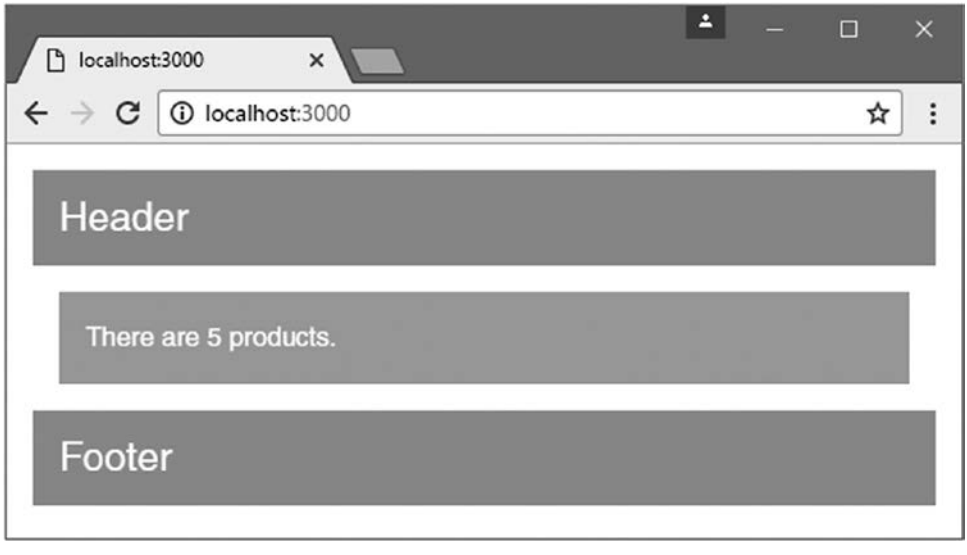


Рис. 13.11. Передача контекстных данных для повторяющегося контекста

КОМПАКТНЫЙ СИНТАКСИС

Директива ngTemplateOutlet может использоваться с более компактным синтаксисом:

```
...
<div *ngTemplateOutlet="titleTemplate"      [ngOutletContext]=
                                            "{title: 'Footer'}">
</div>
...
```

Такое применение директивы не дает особых преимуществ, потому что управляющий элемент удаляется из документа HTML и заменяется контентом элемента template; это означает, что на самом деле неважно, к какому элементу применяется директива.

Ограничения односторонних привязок данных

Выражения, используемые в односторонних привязках и директивах, напоминают код JavaScript, но это не означает, что в них можно использовать все возможности языка JavaScript (или TypeScript). Ограничения и их причины рассматриваются ниже.

Идемпотентные выражения

Односторонние привязки должны быть *идемпотентными*; это означает, что их многократное вычисление не должно изменять состояние приложения. Чтобы продемонстрировать, почему это так, я добавил отладочную команду в метод getProductCount компонента (листинг 13.15).

ПРИМЕЧАНИЕ

Angular поддерживает изменение состояния приложения, но для этого следует использовать методы, описанные в главе 14.

Листинг 13.15. Добавление команды в файл component.ts

```
...
getProductCount(): number {
    console.log("getProductCount invoked");
    return this.getProducts().length;
}
...
```

После сохранения изменений и перезагрузки страницы в браузере на консоли JavaScript браузера появляется длинная серия сообщений:

```
...
getProductCount invoked
getProductCount invoked
getProductCount invoked
getProductCount invoked
...
```

Как показывают сообщения, Angular многократно вычисляет выражение привязки, прежде чем выводить контент в браузере. Если выражение изменяет состояние приложения (например, объект удаляется из очереди), вы не получите ожидаемых результатов в тот момент, когда шаблон отображается для пользователя. Чтобы избежать подобных проблем, Angular ограничивает возможности использования выражений. В листинге 13.16 в компонент добавляется свойство `counter` для демонстрации.

Листинг 13.16. Добавление свойства в файл component.ts

```
import { ApplicationRef, Component } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";

@Component({
    selector: "app",
    templateUrl: "app/template.html"
})
export class ProductComponent {
    model: Model = new Model();

    // ...Конструктор и методы опущены для краткости...

    targetName: string = "Kayak";

    counter: number = 1;
}
```

В листинге 13.17 добавляется привязка, выражение которой увеличивает счетчик при вычислении.

Листинг 13.17. Добавление привязки в файл template.html

```
<template #titleTemplate let-text="title">
    <h4 class="p-a-1 bg-success">{{text}}</h4>
</template>

<template [ngTemplateOutlet]="titleTemplate"
          [ngOutletContext]="{title: 'Header'}">
</template>

<div class="bg-info p-a-1 m-a-1">
    There are {{getProductCount()}} products.
</div>

<template [ngTemplateOutlet]="titleTemplate"
          [ngOutletContext]="{title: 'Footer'}">
</template>

<div class='bg-info p-a-1'>
    Counter: {{counter = counter + 1}}
</div>
```

Когда браузер загрузит страницу, на консоль JavaScript выводится ошибка:

```
EXCEPTION: Template parse errors:
Parser Error: Bindings cannot contain assignments at column 11 in [
        Counter: {{counter = counter + 1}}
] in Products@17:27
```

Angular сообщает об ошибке, если выражение привязки данных содержит оператор, который может использоваться с данными (например, `=`, `+=`, `-+`, `++` и `--`).

Кроме того, когда Angular работает в режиме разработки, выполняется дополнительная проверка того, не изменялись ли односторонние привязки данных после

вычисления их выражений. В листинге 13.18 в компонент добавляется свойство, которое удаляет и возвращает объект `Product` из массива модели.

ПРИМЕЧАНИЕ

О том, как переключаться между режимом разработки (используется по умолчанию) и рабочим режимом, рассказано в главе 10.

Листинг 13.18. Изменение данных в файле component.ts

```
import { ApplicationRef, Component } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";

@Component({
    selector: "app",
    templateUrl: "app/template.html"
})
export class ProductComponent {
    model: Model = new Model();

    // ...Конструктор и методы опущены для краткости...

    counter: number = 1;

    get nextProduct(): Product {
        return this.model.getProducts().shift();
    }
}
```

В листинге 13.19 показана привязка данных, использованная для чтения свойства `nextProduct`.

Листинг 13.19. Привязка к свойству в файле template.html

```
<template #titleTemplate let-text="title">
    <h4 class="p-a-1 bg-success">{{text}}</h4>
</template>

<template [ngTemplateOutlet]="titleTemplate"
          [ngOutletContext]="{title: 'Header'}">
</template>

<div class="bg-info p-a-1 m-a-1">
    There are {{getProductCount()}} products.
</div>

<template [ngTemplateOutlet]="titleTemplate"
          [ngOutletContext]="{title: 'Footer'}">
</template>

<div class='bg-info p-a-1'>
    Next Product is {{nextProduct.name}}
</div>
```

После сохранения изменений и обработки шаблона вы увидите, что при попытке изменить данные приложения в привязке данных на консоль JavaScript выдается следующая ошибка:

```
ORIGINAL EXCEPTION: Expression has changed after it was checked.
Previous value: 'There are 5 products.'.
Current value: 'There are 4 products.'
```

Контекст выражения

Вычисляя выражение, Angular действует в контексте компонента шаблона; это обстоятельство позволяет шаблону обращаться к методам и свойствам без каких-либо префиксов:

```
...
<div class="bg-info p-a-1">
    There are {{getProductCount()}} products.
</div>
...
```

Когда Angular обрабатывает эти выражения, компонент предоставляет метод `getProductCount`; Angular вызывает его с заданными аргументами, а затем встраивает результат в документ HTML. Таким образом, компонент предоставляет *контекст выражения* шаблона.

Термин «контекст выражения» означает, что вы не можете обращаться к объектам, определенным за пределами компонента шаблона; в частности, шаблоны не могут обращаться к глобальному пространству имен. Глобальное пространство имен используется для определения стандартных служебных средств (таких, как объект `console` с методом `log`, который я использовал для вывода отладочной информации на консоль JavaScript в браузере). Также глобальное пространство имен включает объект `Math` с полезными арифметическими методами, такими как `min` и `max`.

Для демонстрации этого ограничения в листинге 13.20 в шаблон добавляется привязка со строковой интерполяцией, использующая метод `Math.floor` для округления числа.

Листинг 13.20. Обращение к глобальному пространству имен из файла template.html

```
<template #titleTemplate let-text="title">
    <h4 class="p-a-1 bg-success">{{text}}</h4>
</template>

<template [ngTemplateOutlet]="titleTemplate"
          [ngOutletContext]="{title: 'Header'}">
</template>

<div class="bg-info p-a-1 m-a-1">
    There are {{getProductCount()}} products.
</div>

<template [ngTemplateOutlet]="titleTemplate"
          [ngOutletContext]="{title: 'Footer'}">
</template>

<div class='bg-info p-a-1'>
    The rounded price is {{Math.floor(getProduct(1).price)}}
</div>
```

Во время обработки шаблона Angular выдает следующую ошибку на консоль JavaScript в браузере:

```
EXCEPTION: TypeError: Cannot read property 'floor' of undefined
```

В сообщении об ошибке глобальное пространство имен явно не упоминается. Angular пытается вычислить выражение в контексте компонента, но не может найти свойство `Math`.

Если вы хотите обратиться к функциональности глобального пространства имен, она должна предоставляться компонентом. В нашем примере компонент может просто определить свойство `Math`, которому присваивается глобальный объект, однако выражения шаблонов должны быть как можно более простыми и ясными, поэтому лучше определить метод, предоставляющий шаблону необходимую функциональность (листинг 13.21).

Листинг 13.21. Определение метода в файле component.ts

```
import { ApplicationRef, Component } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";

@Component({
    selector: "app",
    templateUrl: "app/template.html"
})

export class ProductComponent {
    model: Model = new Model();

    // ...Конструктор и методы опущены для краткости...

    counter: number = 1;

    get nextProduct(): Product {
        return this.model.getProducts().shift();
    }

    getProductPrice(index: number): number {
        return Math.floor(this.getProduct(index).price);
    }
}
```

В листинге 13.22 привязка данных в шаблоне изменена для использования нового метода.

Листинг 13.22. Обращение к глобальному пространству имен в файле template.html

```
<template #titleTemplate let-text="title">
    <h4 class="p-a-1 bg-success">{{text}}</h4>
</template>

<template [ngTemplateOutlet]="titleTemplate"
          [ngOutletContext]="{title: 'Header'}">
</template>
```

```
<div class="bg-info p-a-1 m-a-1">
    There are {{getProductCount()}} products.
</div>

<template [ngTemplateOutlet]="titleTemplate"
          [ngOutletContext]="{title: 'Footer'}">
</template>

<div class='bg-info p-a-1'>
    The rounded price is {{getProductPrice(1)}}
</div>
```

В процессе обработки шаблона Angular вызывает метод `getProductPrice` и неявно использует объект `Math` из глобального пространства имен. Результат показан на рис. 13.12.

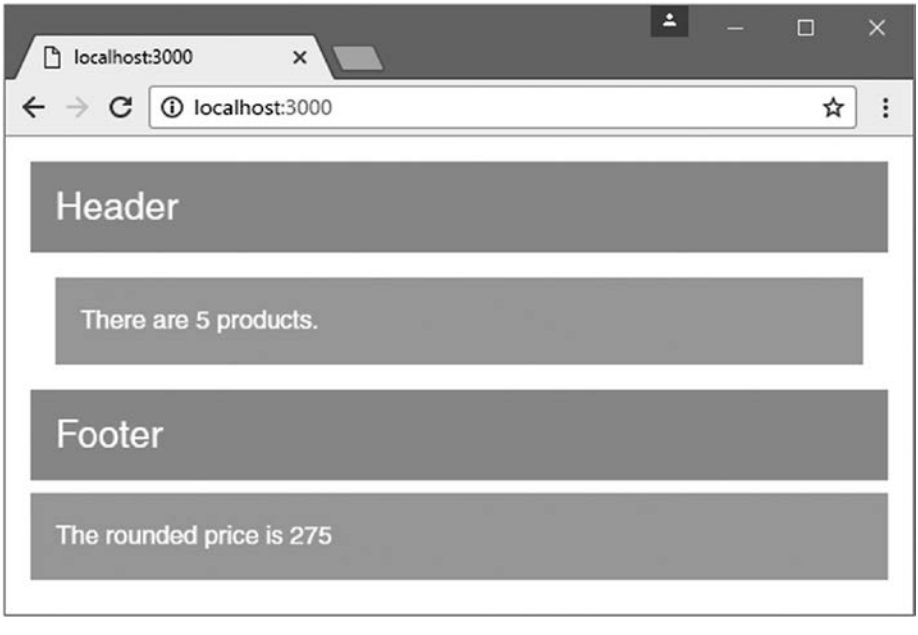


Рис. 13.12. Обращение к функциональности глобального пространства имен

Итоги

Эта глава была посвящена использованию встроенных директив шаблонов. Вы научились выбирать контент при помощи директив `ngIf` и `ngSwitch` и повторять его директивой `ngFor`. Вы узнали, почему некоторые имена директив должны снабжаться префиксом `*`, и рассмотрели ограничения для выражений шаблонов, используемых в этих директивах и с односторонними привязками вообще. В следующей главе я расскажу, как привязки данных используются с событиями и элементами форм.

14 События и формы

В этой главе продолжается рассмотрение базовой функциональности Angular; основное внимание будет уделяться обработке взаимодействий с пользователем. Я расскажу, как создавать привязки событий и как использовать двусторонние привязки для управления потоком данных между моделью и шаблоном. Один из важнейших механизмов взаимодействия с пользователем в веб-приложениях — формы HTML. Я объясню, как события и двусторонние привязки данных обеспечивают их поддержку и проверку данных, введенных пользователем. В табл. 14.1 события и формы представлены в контексте.

Таблица 14.1. Привязки событий и формы в контексте

Вопрос	Ответ
Что это такое?	Привязки событий вычисляют выражение при инициировании события (нажатия клавиши пользователем, перемещения мыши, отправки данных формы и т. д.). На этой основе строится расширенная функциональность форм, автоматически проверяющих введенные данные
Для чего они нужны?	Привязки событий и формы позволяют пользователю изменять состояние приложения с изменением или добавлением данных в модели
Как они используются?	По-разному. За подробностями обращайтесь к примерам
Есть ли у них недостатки или скрытые проблемы?	Как и у всех привязок Angular, главная проблема — неправильный выбор скобок для обозначения привязки. Будьте внимательны с примерами этой главы и перепроверьте синтаксис привязок, если результаты отличаются от ожидаемых
Есть ли альтернативы?	Нет. Привязки событий и формы — неотъемлемая часть приложений Angular

В табл. 14.2 приведена краткая сводка материала главы.

Таблица 14.2. Сводка материала главы

Проблема	Решение	Листинг
Включение поддержки форм	Добавьте модуль @angular/forms в приложение	1–3
Реакция на событие	Используйте привязку события	6–8
Получение подробной информации о событии	Используйте объект $event	9
Обращение к элементам в шаблоне	Определите переменные шаблонов	10
Организация передачи данных в обоих направлениях между элементом и компонентом	Используйте двустороннюю привязку данных	11, 12
Получение ввода от пользователя	Используйте форму HTML	13, 14
Проверка данных, введенных пользователем	Выполните проверку данных формы	15–24
Определение информации проверки в коде JavaScript	Используйте форму на базе модели	25–30
Расширение встроенной функциональности проверки данных форм	Определите нестандартный класс проверки данных формы	31–32

Подготовка проекта

В этой главе мы продолжим использовать проект example, созданный в главе 11 и измененный в следующих главах.

Добавление модуля

Возможности, описанные в этой главе, зависят от модуля форм Angular; этот модуль необходимо добавить в приложение. Начните с включения новой записи в секцию `dependencies` файла package.json (листинг 14.1).

Листинг 14.1. Добавление пакета forms в файл package.json

```
{
  "dependencies": {
    "@angular/common": "2.2.0",
    "@angular/compiler": "2.2.0",
    "@angular/core": "2.2.0",
    "@angular/platform-browser": "2.2.0",
    "@angular/platform-browser-dynamic": "2.2.0",
    "@angular/upgrade": "2.2.0",
    "@angular/forms": "2.2.0",
    "reflect-metadata": "0.1.8",
    "rxjs": "5.0.0-beta.12",
    "zone.js": "0.6.26",
    "core-js": "2.4.1",
```

```
    "classlist.js": "1.1.20150312",
    "systemjs": "0.19.40",
    "bootstrap": "4.0.0-alpha.4"
  },

  "devDependencies": {
      "lite-server": "2.2.2",
      "typescript": "2.0.2",
      "typings": "1.3.2",
      "concurrently": "2.2.0"
  },

  "scripts": {
    "start": "concurrently \"npm run tscwatch\" \"npm run lite\" ",
    "tsc": "tsc",
    "tscwatch": "tsc -w",
    "lite": "lite-server",
    "typings": "typings",
    "postinstall": "typings install"
  }
}
```

Выполните следующую команду из папки example, чтобы загрузить и установить пакет `forms`:

```
npm install
```

Затем обновите конфигурацию пакета SystemJS, чтобы он знал, как найти пакет `forms` при загрузке приложения (листинг 14.2).

Листинг 14.2. Добавление модуля forms в файл systemjs.config.js

```
(function(global) {

    var paths = {
        "rxjs/*": "node_modules/rxjs/bundles/Rx.min.js",
        "@angular/*": "node_modules/@angular/*"
    }

    var packages = { "app":  {} };

    var angularModules = ["common", "compiler",
     "core", "platform-browser", "platform-browser-dynamic",
     "forms"];

    angularModules.forEach(function(pkg) {
        packages["@angular/" + pkg] = {
            main: "/bundles/" + pkg + ".umd.min.js"
        };
    });

    System.config({ paths: paths, packages: packages });

})(this);
```

Обновите модуль Angular: в нем необходимо объявить, что приложение зависит от модуля `forms` (листинг 14.3).

Листинг 14.3. Объявление зависимости в файле app.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { ProductComponent } from "./component";
import { FormsModule} from "@angular/forms";

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

Свойство `imports` декоратора `NgModule` определяет зависимости приложения. Добавление `FormsModule` в список зависимостей активизирует функциональность форм и открывает возможность их использования в приложении.

Подготовка компонента и шаблона

В листинге 14.4 из класса компонента удаляется конструктор и некоторые методы (чтобы упростить код, насколько это возможно).

Листинг 14.4. Упрощение кода компонента в файле component.ts

```
import { ApplicationRef, Component } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";

@Component({
    selector: "app",
    templateUrl: "app/template.html"
})
export class ProductComponent {
    model: Model = new Model();

    getProduct(key: number): Product {
        return this.model.getProduct(key);
    }

    getProducts(): Product[] {
        return this.model.getProducts();
    }
}
```

В листинге 14.5 упрощается шаблон компонента. В нем остается только таблица, заполненная директивой `ngFor`.

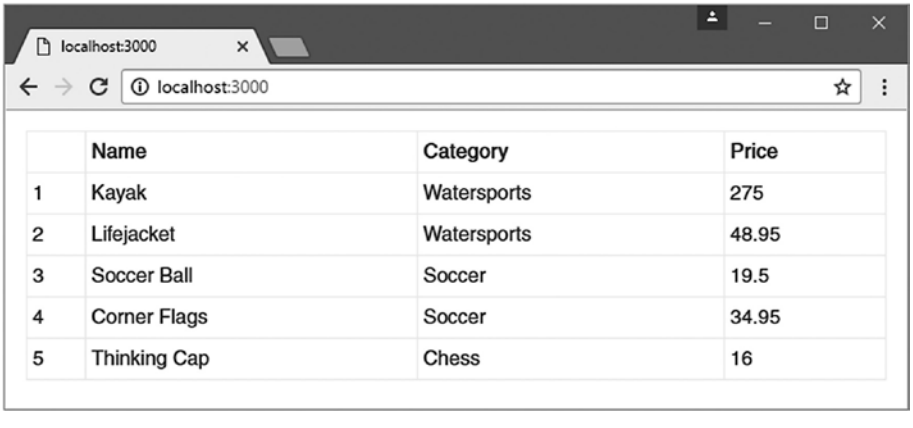
Листинг 14.5. Упрощение шаблона в файле template.html

```
<table class="table table-sm table-bordered m-t-1">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
    <tr *ngFor="let item of getProducts(); let i = index">
        <td>{{i + 1}}</td>
        <td>{{item.name}}</td>
        <td>{{item.category}}</td>
        <td>{{item.price}}</td>
    </tr>
</table>
```

Чтобы запустить сервер для разработки, откройте окно командной строки, перейдите в папку example и выполните следующую команду:

```
npm start
```

На экране появляется новое окно (или вкладка) браузера, в котором отображается таблица (рис. 14.1).



	Name	Category	Price
1	Kayak	Watersports	275
2	Lifejacket	Watersports	48.95
3	Soccer Ball	Soccer	19.5
4	Corner Flags	Soccer	34.95
5	Thinking Cap	Chess	16

Рис. 14.1. Запуск приложения

Использование привязки события

Привязка события используется для обработки событий, отправленных управляющим элементом. В листинге 14.6 продемонстрирована привязка события, при помощи которой пользователь взаимодействует с приложением Angular.

Листинг 14.6. Использование привязки события в файле template.html

```
<div class="bg-info p-a-1">
    Selected Product: {{selectedProduct || '(None)'}}
</div>
<table class="table table-sm table-bordered m-t-1">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
    <tr *ngFor="let item of getProducts(); let i = index">
```

```
        <td (mouseover)="selectedProduct=item.name">{{i + 1}}</td>
        <td>{{item.name}}</td>
        <td>{{item.category}}</td>
        <td>{{item.price}}</td>
    </tr>
</table>
```

Сохранив изменения в шаблоне, протестируйте привязку — наведите указатель мыши на первый столбец с серией цифр. При перемещении мыши от строки к строке название товара этой строки выводится в верхней части страницы (рис. 14.2).

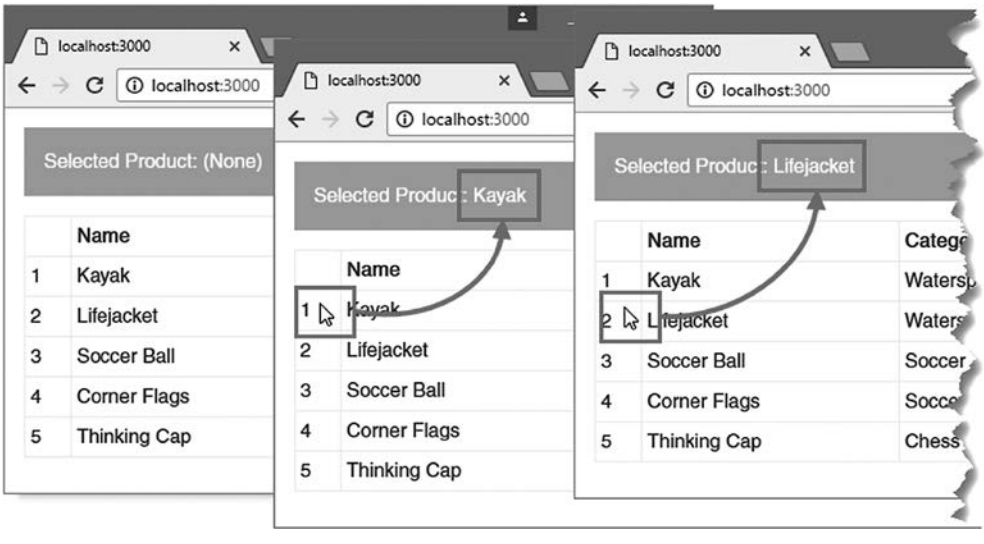


Рис. 14.2. Использование привязки события

Этот простой пример демонстрирует структуру привязки события, изображенную на рис. 14.3.

Рис. 14.3. Строение привязки события

Привязка события состоит из четырех частей:

- *Управляющий элемент* является источником событий для привязки.
- *Круглые скобки* сообщают Angular, что определяется привязка события — разновидность односторонней привязки, в которой данные передаются от элемента к коду приложения.
- *Событие* определяет, для какого события определяется привязка.
- *Выражение* вычисляется при инициировании события.

Обращаясь к привязке в листинге 14.6, вы видите, что управляющим элементом является элемент `td`; это означает, что данный элемент становится источником событий. В привязке указано событие `mouseover`, которое инициируется, когда указатель мыши наводится на часть экрана, занятую управляющим элементом.

В отличие от односторонних привязок, выражения в привязках событий могут вносить изменения в состояние приложения и могут включать операторы присваивания (например, `=`). Выражение привязки присваивает переменной `selectedProduct` значение `item.name`. Переменная `selectedProduct` используется в привязке со строковой интерполяцией в начале шаблона:

```
...
<div class="bg-info p-a-1">
    Selected Product: {{selectedProduct || '(None)'}}
</div>
...
```

Обратите внимание: привязка со строковой интерполяцией обновляется, когда значение переменной `selectedProduct` изменяется привязкой события. Вручную запускать процесс обнаружения изменений методом `ApplicationRef.tick` уже не нужно, потому что привязки и директивы этой главы решают эту задачу автоматически.

РАБОТА С СОБЫТИЯМИ DOM

Если вы еще не знакомы с событиями, которые может отправлять элемент HTML, очень хорошая сводка доступна по адресу developer.mozilla.org/en-US/docs/Web/Events. Впрочем, таких событий довольно много, и не все они достаточно широко или последовательно поддерживаются всеми браузерами. Начните с разделов «события DOM» или «события HTML DOM» на странице mozilla.org; в них определяются основные взаимодействия пользователя с элементом (щелчки, перемещение указателя, отправка данных форм и т. д.), которые заведомо работают практически во всех браузерах.

Если вы используете более редкие события, убедитесь в том, что они доступны в ваших целевых браузерах и работают именно так, как ожидается. Превосходный сайт http://caniuse.com предоставляет подробную информацию о том, какие возможности реализованы теми или иными браузерами, но вы также должны провести тщательное тестирование.

Динамически определяемые свойства

Возможно, вас интересует, откуда взялась переменная `selectedProduct` из листинга 14.6 — ведь она используется в шаблоне без предварительного определения в компоненте? Переменная была создана при первом срабатывании привязки со-

бытия, потому что JavaScript разрешает определять события для объектов динамически.

Объектом в данном случае является компонент. Этот пример напоминает о том, что хотя TypeScript формирует структуру при написании кода, приложение Angular во время выполнения содержит простой код JavaScript. Это означает, что вы можете пользоваться динамическими средствами JavaScript, когда это удобно. С другой стороны, вы рискуете получить непредвиденные результаты, поэтому действовать следует осторожно.

И хотя свойства могут определяться динамически, безопаснее определить все свойства, используемые шаблоном, в компоненте; а если вы пишете другие методы и свойства, которые от них зависят, вам придется это сделать, иначе компилятор TypeScript выдаст сообщение об ошибке. В листинге 14.7 компонент обновляется для отслеживания выбранного товара; это означает определение свойства `selectedProduct`, которое ранее создавалось динамически.

Листинг 14.7. Расширение компонента в файле component.ts

```
import { ApplicationRef, Component } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";

@Component({
    selector: "app",
    templateUrl: "app/template.html"
})
export class ProductComponent {
    model: Model = new Model();

    getProduct(key: number): Product {
        return this.model.getProduct(key);
    }

    getProducts(): Product[] {
        return this.model.getProducts();
    }

    selectedProduct: string;

    getSelected(product: Product): boolean {
        return product.name == this.selectedProduct;
    }
}
```

Кроме свойства `selectedProduct` появился новый метод `getSelected`, который получает объект `Product` и сравнивает его название со свойством `selectedProduct`. В листинге 14.8 метод `getSelected` используется привязкой класса для управления принадлежностью к классу `bg-info` — классу Bootstrap, который назначает цвет фона элемента.

Листинг 14.8. Управление принадлежностью к классу в файле template.html

```
<div class="bg-info p-a-1">
    Selected Product: {{selectedProduct || '(None)'}}
</div>
<table class="table table-sm table-bordered m-t-1">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
    <tr *ngFor="let item of getProducts(); let i = index"
            [class.bg-info]="getSelected(item)">
        <td (mouseover)="selectedProduct=item.name">{{i + 1}}</td>
        <td>{{item.name}}</td>
        <td>{{item.category}}</td>
        <td>{{item.price}}</td>
    </tr>
</table>
```

В результате элементы `tr` добавляются в класс `bg-info`, когда значение свойства `selectedProduct` совпадает со свойством `name` объекта `Product`, использованного для их создания; изменение вносится привязкой события при срабатывании события `mouseover` (рис. 14.4).

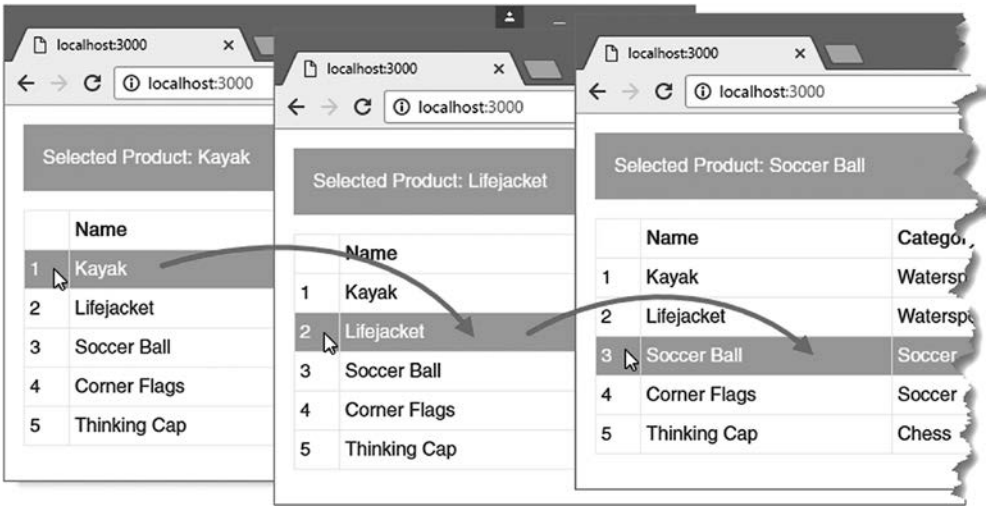


Рис. 14.4. Выделение строк таблицы через привязку события

Пример показывает, как взаимодействие с пользователем передает новые данные приложению и запускает процесс обнаружения изменений, что заставляет Angular заново вычислять выражения, используемые в строковой интерполяции и привязке классов. Именно этот механизм передачи данных вдыхает жизнь в приложения Angular: привязки и директивы, описанные в главах 12 и 13, динамически реагируют на изменения в состоянии приложения и создают контент, который генерируется и находится под полным управлением браузера.

Использование данных события

В предыдущем примере директива события используется для объединения двух блоков данных, предоставленных компонентом: когда инициируется событие `mouseover`, привязка задает значение свойства `selectedProduct` компонента с использованием значения данных, переданного директиве `ngFor` из метода `getProducts` компонента.

Привязка события также может использоваться для передачи приложению новых данных от самого события. В листинге 14.9 в шаблон добавляется элемент `input`, а привязка события используется для прослушивания события `input`, которое инициируется при изменении контента элемента `input`.

Листинг 14.9. Использование объекта события в файле template.html

```
<div class="bg-info p-a-1">
    Selected Product: {{selectedProduct || '(None)'}}
</div>
<table class="table table-sm table-bordered m-t-1">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
    <tr *ngFor="let item of getProducts(); let i = index"
            [class.bg-info]="getSelected(item)">
        <td (mouseover)="selectedProduct=item.name">{{i + 1}}</td>
        <td>{{item.name}}</td>
        <td>{{item.category}}</td>
        <td>{{item.price}}</td>
    </tr>
</table>
<div class="form-group">
    <label>Product Name</label>
    <input class="form-control" (input)="selectedProduct=$event.target.value" />
</div>
```

Когда браузер инициирует событие, он предоставляет объект с описанием этого события. Существуют разные типы объектов для разных категорий событий (события мыши, события клавиатуры, события форм и т. д.), но все события содержат три свойства, описанные в табл. 14.3.

Таблица 14.3. Свойства, общие для всех объектов событий DOM

Имя	Описание
type	Свойство возвращает строку, идентифицирующую тип инициированного события
target	Свойство возвращает объект, инициирующий событие; обычно это объект, представляющий элемент HTML в DOM
timeStamp	Свойство возвращает число с временной меткой момента инициирования события (в миллисекундах с 1 января 1970 года)

Объект события присваивается переменной шаблона с именем `$event`, а новое выражение привязки в листинге 14.9 использует переменную для обращения к свойству `target` объекта события.

Элемент `input` представляется в модели DOM объектом `HTMLInputElement`, который определяет свойство `value`, используемое для чтения и записи содержимого элемента `input`.

Выражение привязки обрабатывает событие `input`, задавая свойству `selectedProduct` компонента значение свойства `value` элемента `input`:

```
...
<input class="form-control" (input)="selectedProduct=$event.target.value" />
...
```

Событие `input` инициируется при изменении содержимого элемента `input` пользователем, поэтому свойство `selectedProduct` компонента обновляется содержимым элемента `input` после каждого нажатия клавиши. В процессе ввода текста в элементе `input` введенный текст отображается в верхней части окна браузера, при этом используется привязка со строковой интерполяцией.

Привязка `ngClass`, примененная к элементам `tr`, назначает цвет фона строк таблицы в том случае, если свойство `selectedProduct` совпадает с именем представленного продукта. И теперь, когда значение свойства `selectedProduct` определяется контентом элемента `input`, при вводе названия товара соответствующая строка выделяется, как показано на рис. 14.5.

Организация совместной работы разных привязок занимает центральное место в эффективной разработке приложений Angular. Она позволяет создавать приложения, немедленно реагирующие на действия пользователя и изменения в модели данных.

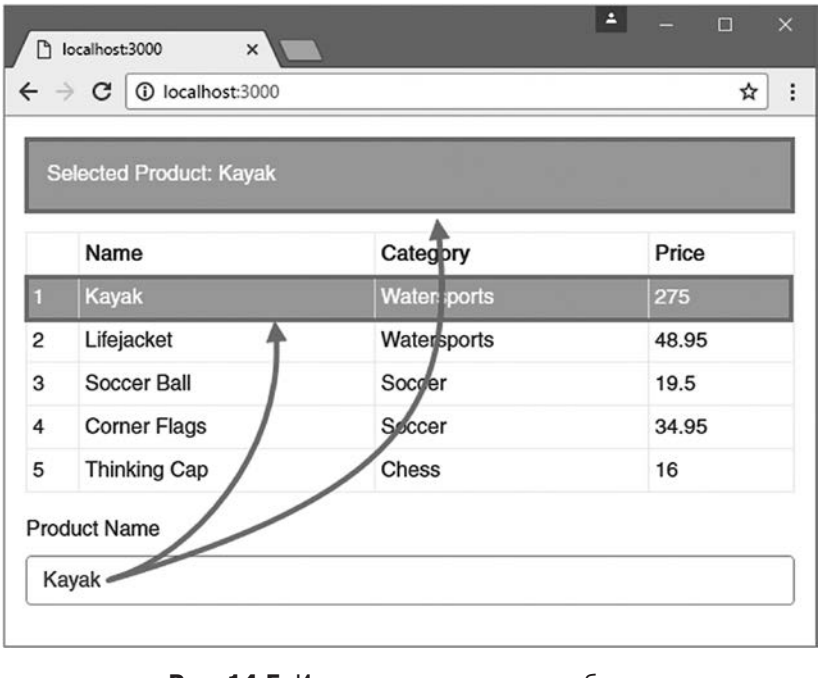


Рис. 14.5. Использование данных события

Использование ссылочных переменных шаблона

В главе 13 было показано, как использовать переменные шаблонов для передачи данных в шаблоне, — например, как определить переменную для текущего объекта при использовании директивы `ngFor`. Ссылочные переменные шаблонов — разновидность переменных шаблонов, которая может использоваться для ссылок на элементы внутри шаблона (листинг 14.10).

Листинг 14.10. Использование переменной шаблона в файле template.html

```
<div class="bg-info p-a-1">
    Selected Product: {{product.value || '(None)'}}
</div>
<table class="table table-sm table-bordered m-t-1">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
    <tr *ngFor="let item of getProducts(); let i = index"
            (mouseover)="product.value=item.name"
            [class.bg-info]="product.value==item.name">
        <td>{{i + 1}}</td>
        <td>{{item.name}}</td>
        <td>{{item.category}}</td>
        <td>{{item.price}}</td>
    </tr>
</table>
<div class="form-group">
    <label>Product Name</label>
    <input #product class="form-control" (input)="false" />
</div>
```

Ссылочные переменные определяются символом `#`, за которым следует имя переменной. В листинге переменная с именем `product` определяется следующим образом:

```
...
<input #product class="form-control" (input)="false" />
...
```

Обнаруживая ссылочную переменную в шаблоне, Angular присваивает ей элемент, в котором она была применена. Например, в данном примере ссылочной переменной `product` присваивается объект, представляющий элемент `input` в DOM (объект `HTMLInputElement`). Ссылочные переменные могут использоваться другими привязками в том же шаблоне. Эта возможность продемонстрирована в привязке со строковой интерполяцией, которая также использует переменную `product`:

```
...
Selected Product: {{product.value || '(None)'}}
...
```

Привязка выводит свойство `value`, определяемое объектом `HTMLInputElement`, который был присвоен переменной `product`, или строку `(None)`, если свойство `value` возвращает `null`. Переменные шаблонов также могут использоваться для изменения состояния элемента, как в следующей привязке:

```
...
<tr *ngFor="let item of getProducts(); let i = index"
    (mouseover)="product.value=item.name"
    [class.bg-info]="product.value==item.name">
...
```

Привязка события реагирует на событие `mouseover`, задавая значение свойства `value` объекта `HTMLInputElement`, присвоенного переменной `product`. В результате при наведении указателя мыши на один из элементов `tr` изменяется текст в элементе `input`.

У примера с привязкой события `input` для элемента `input` есть одна небольшая странность:

```
...
<input #product class="form-control" (input)="false" />
...
```

Angular не обновляет привязки данных в шаблоне, когда пользователь редактирует содержимое элемента `input`, при отсутствии привязки события для этого элемента. Назначение `false` в привязке предоставляет Angular выражение для обработки, чтобы начался процесс обновления, а текущее содержимое элемента `input` распространилось в шаблоне. Впрочем, эта странность связана с чрезмерным расширением роли ссылочной переменной шаблона; в большинстве реальных проектов вам ничего такого делать не придется. Как будет показано ниже (и в следующих главах), большинство привязок данных зависит от переменных, определяемых компонентом шаблона.

ФИЛЬТРАЦИЯ СОБЫТИЙ КЛАВИАТУРЫ

Событие input инициируется при каждом изменении содержимого элемента input. В этом случае вы получаете непосредственную и быструю реакцию на изменения, но столь частые обновления нужны не в каждом приложении (особенно если в состоянии приложения задействованы затратные операции).

В привязках событий реализована встроенная возможность настройки избирательности для событий клавиатуры; это означает, что обновления будут выполняться только при нажатии конкретной клавиши. Следующая привязка реагирует на нажатие любой клавиши:

```
...
<input #product class="form-control" (keyup)="selectedProduct=product.value" />
...
```

Событие keyup принадлежит к числу стандартных событий DOM. В результате приложение будет обновляться при отпускании каждой клавиши во время ввода в элементе input. Также можно более точно указать нужную клавишу, указав ее имя в составе привязки события:

```
...
<input #product class="form-control" (keyup.enter)=
                      "selectedProduct=product.value" />
...
```

Клавиша, на которую реагирует привязка, задается через точку после имени события DOM. Эта привязка определяется для клавиши Enter; в результате изменения в элементе input будут донесены до остальных частей приложения только после нажатия этой клавиши.

Двусторонние привязки данных

Объединяя привязки, можно организовать двустороннюю передачу данных от одного элемента; документ HTML реагирует на изменение модели приложения, а приложение реагирует на отправку события элементом (листинг 14.11).

Листинг 14.11. Создание двусторонней привязки в файле template.html

```
<div class="bg-info p-a-1">
    Selected Product: {{selectedProduct || '(None)'}}
</div>
<table class="table table-sm table-bordered m-t-1">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
    <tr *ngFor="let item of getProducts(); let i = index"
            [class.bg-info]="getSelected(item)">
        <td (mouseover)="selectedProduct=item.name">{{i + 1}}</td>
        <td>{{item.name}}</td>
        <td>{{item.category}}</td>
        <td>{{item.price}}</td>
    </tr>
</table>

<div class="form-group">
    <label>Product Name</label>
    <input class="form-control" (input)="selectedProduct=$event.target.value"
        [value]="selectedProduct || ''" />
</div>

<div class="form-group">
    <label>Product Name</label>
    <input class="form-control" (input)="selectedProduct=$event.target.value"
        [value]="selectedProduct || ''" />
</div>
```

Каждый из элементов `input` имеет две привязки: привязку события и привязку свойства. Привязка события реагирует на событие `input` обновлением свойства `selectedProduct` компонента. Привязка свойства связывает значение свойства `selectedProduct` со свойством `value` элемента.

В результате содержимое двух элементов `input` синхронизируется, а редактирование одного элемента приводит к обновлению другого. Кроме того, так как в шаблоне есть другие привязки, зависящие от свойства `selectedProduct`, редактирование содержимого элемента `input` также изменяет данные, отображаемые привязкой со строковой интерполяцией, и изменяет выделенную строку таблицы (рис. 14.6).

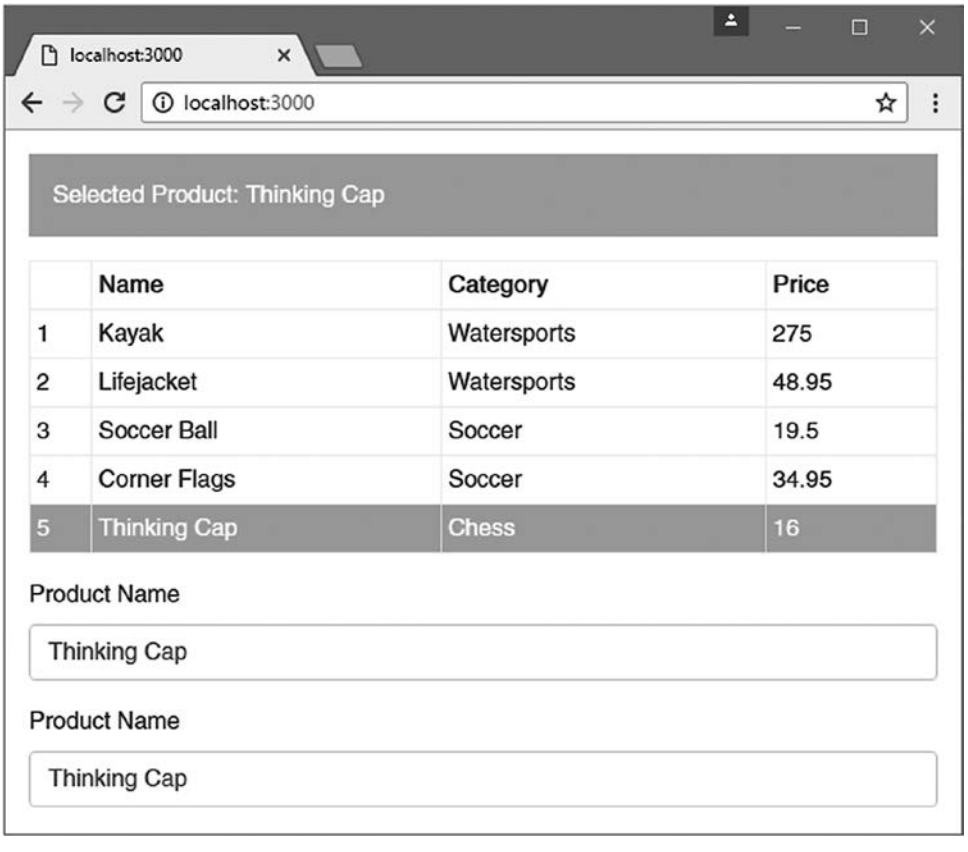


Рис. 14.6. Создание двусторонней привязки данных

Чтобы этот пример стал действительно понятным, поэкспериментируйте с ним в браузере. Введите текст в одном из элементов `input`, и вы увидите, как этот же текст появляется в другом элементе `input` и в элементе `div`, контентом которого управляет привязка со строковой интерполяцией. Если ввести в одном из элементов `input` название товара (например, `Kayak` или `Lifejacket`), вы увидите, что соответствующая строка таблицы выделяется.

Привязка события для `mouseover` остается активной; это означает, что как только вы наведете указатель мыши на первую строку таблицы, из-за изменения значения `selectedProduct` в элементах `input` появится название товара.

Директива ngModel

Директива `ngModel` упрощает двусторонние привязки, чтобы вам не приходилось применять как привязку события, так и привязку свойства к одному элементу. В листинге 14.12 показано, как заменить разные привязки директивой `ngModel`.

Листинг 14.12. Использование директивы ngModel в файле template.html

```
<div class="bg-info p-a-1">
    Selected Product: {{selectedProduct || '(None)'}}
</div>
<table class="table table-sm table-bordered m-t-1">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
    <tr *ngFor="let item of getProducts(); let i = index"
            [class.bg-info]="getSelected(item)">
        <td (mouseover)="selectedProduct=item.name">{{i + 1}}</td>
        <td>{{item.name}}</td>
        <td>{{item.category}}</td>
        <td>{{item.price}}</td>
    </tr>
</table>

<div class="form-group">
    <label>Product Name</label>
    <input class="form-control" [(ngModel)]="selectedProduct" />
</div>

<div class="form-group">
    <label>Product Name</label>
    <input class="form-control" [(ngModel)]="selectedProduct" />
</div>
```

Директива `ngModel` объединяет синтаксис привязок свойств и событий (рис. 14.7).

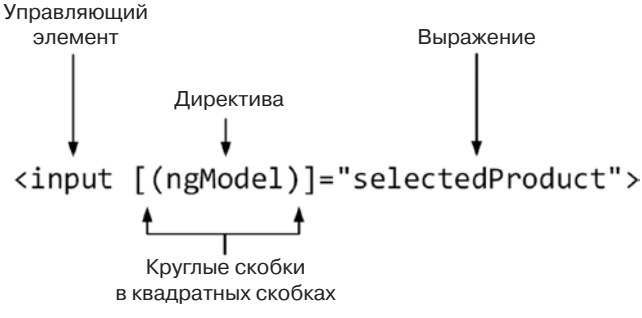


Рис. 14.7. Строение двусторонней привязки данных

Комбинация из квадратных и круглых скобок обозначает двустороннюю привязку данных; круглые скобки должны находиться внутри квадратных: `[(` и `)]`. Группа разработки Angular называет этот синтаксис привязки «бананом в коробке», потому что за скобками `[()]` можно увидеть именно такую картинку (если очень постараться).

Целью привязки является директива `ngModel`, которая включена в Angular для упрощения создания двусторонних привязок данных для элементов форм (таких, как элементы `input` из нашего примера).

Выражение двусторонней привязки определяет имя свойства, которое используется для задания отдельных привязок во внутренней реализации. При изменении содер-

жимого элемента `input` новое содержимое будет использовано для обновления значения свойства `selectedProduct`. И наоборот, при изменении свойства `selectedProduct` значение будет использовано для обновления содержимого элемента.

Директива `ngModel` знает комбинацию свойств и событий, определяемую стандартными элементами HTML. При этом «за кулисами» привязка события применяется к событию `input`, а привязка свойства применяется к свойству `value`.

ПРИМЕЧАНИЕ

Помните, что с привязкой ngModel должны использоваться как квадратные, так и круглые скобки. Если поставить только круглые скобки — (ngModel), то вы создадите привязку события для события с именем ngModel, а такое событие не существует. В результате элемент не будет обновлен или не обновится другая часть приложения. Директива ngModel может использоваться только с квадратными скобками — [ngModel]; Angular задаст начальное значение элемента, но не будет прослушивать события, а это означает, что внесенные пользователем изменения не будут автоматически отражаться в модели приложения.

Работа с формами

Многие веб-приложения используют формы для получения данных от пользователя. Двусторонняя привязка ngModel, описанная выше, закладывает основу для создания простых форм, которые могут дополняться расширенными возможностями. В этом разделе мы создадим форму для создания новых товаров и включения их в модель данных приложения. Далее будут рассмотрены некоторые расширенные возможности форм, предоставляемые Angular.

Добавление формы в приложение

В листинге 14.13 приведены изменения компонента, которые будут использоваться при создании формы. Из приложения также удаляются некоторые функции, которые стали лишними.

Листинг 14.13. Изменение компонента в файле component.ts

```
import { ApplicationRef, Component } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";

@Component({
    selector: "app",
    templateUrl: "app/template.html"
})
export class ProductComponent {
    model: Model = new Model();

    getProduct(key: number): Product {
        return this.model.getProduct(key);
```

```
    }

    getProducts(): Product[] {
        return this.model.getProducts();
    }

    newProduct: Product = new Product();

    get jsonProduct() {
        return JSON.stringify(this.newProduct);
    }

    addProduct(p: Product) {
        console.log("New Product: " + this.jsonProduct);
    }
}
```

В листинге добавляется новое свойство с именем `newProduct` для хранения данных, введенных на форме пользователем. Также имеется свойство `jsonProduct` с get-методом, который возвращает представление свойства `newProduct` в формате JSON; оно будет использоваться в шаблоне для демонстрации эффекта двусторонних привязок. (Я не могу создать JSON-представление прямо в шаблоне, потому что объект `JSON` определяется в глобальном пространстве имен, к которому, как рассказано в главе 13, нельзя обращаться из выражений шаблонов.)

Последнее дополнение — метод `addProduct`, который выводит значение метода `jsonProduct` на консоль; это позволит мне продемонстрировать некоторые возможности, связанные с формами, перед добавлением поддержки обновления модели данных позднее в этой главе.

В листинге 14.14 существующий контент шаблона заменяется серией элементов `input` для каждого из свойств, определяемых классом `Product`.

Листинг 14.14. Добавление элементов input в файле template.html

```
<div class="bg-info p-a-1 m-b-1">Model Data: {{jsonProduct}}</div>

<div class="form-group">
    <label>Name</label>
    <input class="form-control" [(ngModel)]="newProduct.name" />
</div>
<div class="form-group">
    <label>Category</label>
    <input class="form-control" [(ngModel)]="newProduct.category" />
</div>
<div class="form-group">
    <label>Price</label>
    <input class="form-control" [(ngModel)]="newProduct.price" />
</div>
<button class="btn btn-primary" (click)="addProduct(newProduct)">Create</button>
```

Каждый элемент `input` группируется с `label` и заключается в элемент `div`, которому назначается стилевое оформление при помощи класса Bootstrap `form-group`.

Отдельным элементам `input` назначается класс Bootstrap `form-control` для управления макетом и стилем.

Привязка `ngModel` применяется к каждому элементу `input` для создания двусторонней привязки с соответствующим свойством объекта `newProduct` компонента:

```
...
<input class="form-control" [(ngModel)]="newProduct.name" />
...
```

Также присутствует элемент `button` с привязкой события `click`, которая вызывает метод `addProduct` компонента с передачей значения `newProduct` в аргументе.

```
...
<button class="btn btn-primary" (click)="addProduct(newProduct)">Create</button>
...
```

Наконец, привязка со строковой интерполяцией используется для отображения свойства `newProduct` в формате JSON в верхней части шаблона:

```
...
<div class="bg-info p-a-1 m-b-1">Model Data: {{jsonProduct}}</div>
...
```

Результат показан на рис. 14.8: это набор элементов `input`, которые обновляют свойства объекта `Product`, находящегося под управлением компонента; изменения немедленно отражаются в данных JSON.

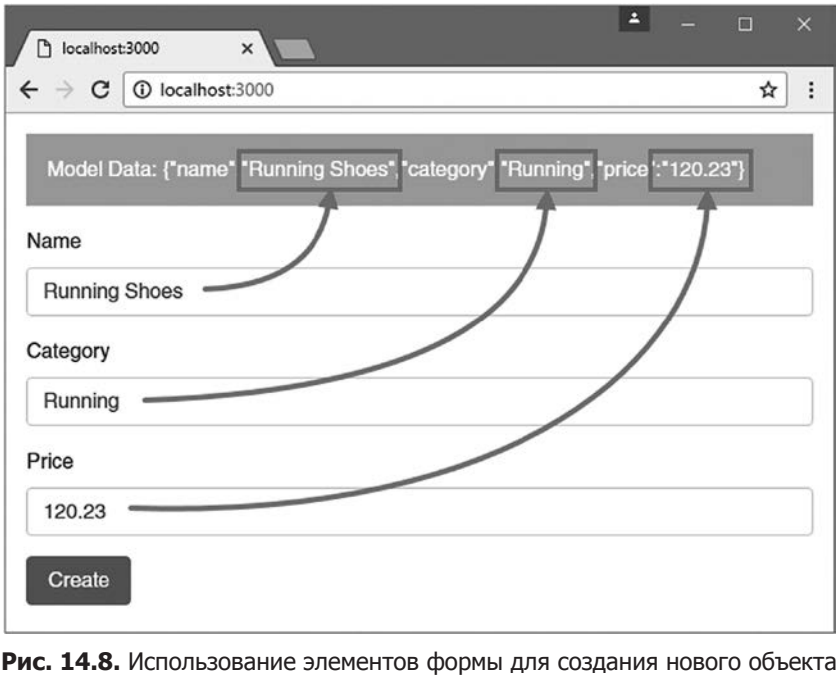


Рис. 14.8. Использование элементов формы для создания нового объекта в модели данных

При нажатии кнопки `Create` представление свойства `newProduct` компонента в формате JSON выводится на консоль JavaScript в браузере. Результат выглядит так:

```
New Product: {"name":"Running Shoes","category":"Running","price":"120.23"}
```

Проверка данных форм

На данный момент в элементах `input` могут вводиться произвольные данные. Проверка данных играет очень важную роль в веб-приложениях, потому что пользователи порой вводят совершенно неожиданные значения — иногда по ошибке, иногда потому, что они хотят завершить процесс как можно быстрее и набирают всякий «мусор», чтобы продвинуться дальше.

Angular предоставляет расширяемую систему проверки содержимого элементов форм на основании подхода, используемого стандартом HTML5. Есть четыре атрибута, которые могут добавляться к элементам `input`; каждый из них определяет правило проверки (табл. 14.4).

Таблица 14.4. Встроенные атрибуты проверки данных Angular

Атрибут	Описание
required	Атрибут используется для задания обязательных значений
minlength	Атрибут используется для задания минимального количества символов
maxlength	Атрибут используется для задания максимального количества символов. Этот тип проверки не может применяться напрямую к элементам форм, потому что он конфликтует с одноименным атрибутом HTML5. Он может использоваться с формами на базе модели (см. далее в этой главе)
pattern	Атрибут используется для задания регулярных выражений, которому должно соответствовать значение, введенное пользователем

Возможно, эти атрибуты будут вам знакомы, потому что они являются частью спецификации HTML, но Angular расширяет их дополнительными возможностями. В листинге 14.15 удаляются все элементы `input`, кроме одного, чтобы по возможности просто продемонстрировать организацию проверки данных в формах. (Удаленные элементы будут восстановлены позже в этой главе.)

Листинг 14.15. Добавление проверки данных в файле template.html

```
<div class="bg-info p-a-1 m-b-1">Model Data: {{jsonProduct}}</div>

<form novalidate (ngSubmit)="addProduct(newProduct)">
    <div class="form-group">
        <label>Name</label>
        <input class="form-control"
            name="name"
            [(ngModel)]="newProduct.name"
            required
            minlength="5"
            pattern="^[A-Za-z ]+$"/>
    </div>
```

```
    <button class="btn btn-primary" type="submit">
        Create
    </button>
</form>
```

Angular требует, чтобы проверяемые элементы определяли атрибут `name`, который используется для идентификации элемента в системе проверки. Так как элемент `input` используется для сохранения значения свойства `Product.name`, атрибуту `name` элемента присваивается значение `name`.

В листинге к элементу `input` также добавляются три из четырех атрибутов проверки данных. Атрибут `required` указывает, что пользователь должен ввести значение; атрибут `minlength` указывает, что значение должно содержать не менее трех символов, а атрибут `pattern` — что допустимы только алфавитные символы и пробелы.

Атрибуты проверки, используемые Angular, совпадают с атрибутами из спецификации HTML5. По этой причине я добавил в элемент `form` атрибут `novalidate`, который запрещает браузеру использовать встроенные средства проверки данных, которые непоследовательно реализуются в разных браузерах и обычно только мешают. Так как проверку данных будет обеспечивать Angular, встроенная реализация этих функций в браузере не нужна.

Наконец, обратите внимание на то, что в шаблон был добавлен элемент `form`. И хотя вы можете использовать элементы `input` независимо, средства проверки данных Angular работают только при наличии элемента `form`, и если вы примените директиву `ngControl` к элементу, не содержащемуся в форме, Angular выдаст сообщение об ошибке. При использовании элемента `form` привязка события обычно используется для специального события с именем `ngSubmit`:

```
...
<form novalidate (ngSubmit)="addProduct(newProduct)">
...
```

Привязка `ngSubmit` обрабатывает событие `submit` элемента `form`. При желании того же эффекта можно добиться привязкой события `click` к отдельным элементам `button` внутри `form`.

Стилевое оформление элементов с использованием классов проверки данных

После того как вы сохраните изменения в шаблоне из листинга 14.15, а браузер перезагрузит HTML, щелкните правой кнопкой мыши на элементе `input` в окне браузера и выберите команду Inspect или Inspect Element в контекстном меню. Браузер выведет представление элемента в формате HTML в окне Developer Tools, и вы увидите, что элементу `input` назначены три класса:

```
...
<input class="form-control ng-pristine ng-invalid ng-touched" minlength="5"
    name="name" pattern="^[A-Za-z ]+$" required="" ng-reflect-name="name">
...
```

Классы, назначаемые элементу `input`, предоставляют подробную информацию о состоянии проверки данных. Существуют три пары классов проверки данных, описанные в табл. 14.5. Элементы всегда относятся к одному классу из каждой пары (итого три класса). Одни и те же классы применяются к элементу `form` для представления общего статуса проверки данных всех содержащихся в них элементов. С изменением статуса элемента `input` директива `ngControl` автоматически переключает классы как для отдельных элементов, так и для элемента `form`.

Таблица 14.5. Классы проверки данных форм Angular

Класс	Описание
ng-untouched ng-touched	Элементу назначается класс ng-untouched, если он не посещался пользователем (что обычно происходит при переходе между полями формы). Когда пользователь посетит элемент, ему назначается класс ng-touched
ng-pristine ng-dirty	Элементу назначается класс ng-pristine, если его содержимое не изменялось пользователем; в противном случае элемент относится к классу ng-dirty. После того как содержимое будет отредактировано, элемент остается в классе ng-dirty, даже если пользователь вернется к предыдущему содержимому
ng-valid ng-invalid	Элементу назначается класс ng-valid, если его содержимое соответствует критериям, определенным назначенными ему правилами проверки данных; в противном случае элемент относится к классу ng-invalid

Эти классы могут использоваться для стилевого оформления элементов форм, чтобы предоставить обратную связь пользователю. В листинге 14.16 в шаблон добавляется элемент `style` и определяются стили, которые сообщают, что пользователь ввел действительные и недействительные данные.

ПРИМЕЧАНИЕ

В реальных приложениях стили обычно определяются в отдельных таблицах стилей и включаются в приложение через файл index.html или с использованием декораторов компонента (см. главу 17). Для простоты я включил стили прямо в шаблон, однако такой подход усложняет сопровождение кода реальных приложений — с ним сложнее понять, откуда взялся тот или иной стиль, при использовании нескольких шаблонов.

Листинг 14.16. Организация обратной связи проверки данных в файле template.html

```
<style>
    input.ng-dirty.ng-invalid { border: 2px solid #ff0000 }
    input.ng-dirty.ng-valid { border: 2px solid #6bc502 }
</style>

<div class="bg-info p-a-1 m-b-1">Model Data: {{jsonProduct}}</div>
```

```
<form novalidate (ngSubmit)="addProduct(newProduct)">
    <div class="form-group">
        <label>Name</label>
        <input class="form-control"
            name="name"
            [(ngModel)]="newProduct.name"
            required
            minlength="5"
            pattern="^[A-Za-z ]+$"/>
    </div>
    <button class="btn btn-primary" type="submit">
        Create
    </button>
</form>
```

Эти стили определяют зеленые и красные рамки для элементов `input`, содержимое которых было изменено и действительно (а следовательно, эти элементы принадлежат классам `ng-dirty` и `ng-valid`) и содержимое которых недействительно (а следовательно, эти элементы принадлежат классам `ng-dirty` и `ng-invalid`). Использование класса `ng-dirty` означает, что внешний вид элементов не изменится после того, как пользователь ввел контент.

Angular проверяет контент и изменяет принадлежность к классам элементов `input` после каждого нажатия клавиши или передачи фокуса. Браузер обнаруживает изменения в элементах и назначает стили динамически, в результате чего пользователи получают обратную связь при вводе данных на форме (рис. 14.9).

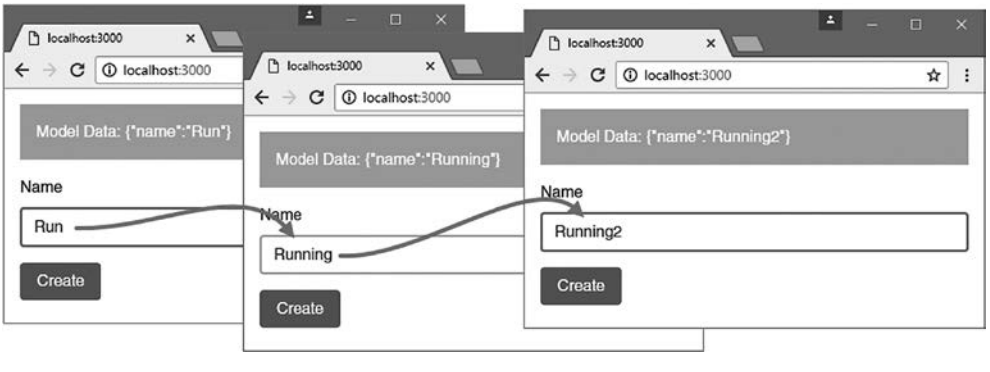


Рис. 14.9. Обратная связь проверки данных

Когда я начинаю вводить данные, элемент `input` помечается как недействительный, потому что количество символов недостаточно для удовлетворения атрибута `minlength`. Когда количество символов достигнет пяти, рамка окрашивается в зеленый цвет, сообщая о том, что данные действительны. Но когда вводится символ 2, рамка снова окрашивается в красный цвет, потому что атрибут `pattern` допускает только буквы и пробелы.

ПРИМЕЧАНИЕ

Если вы взглянете на данные JSON в верхней части рис. 14.9, то увидите, что привязки данных продолжают обновляться, даже если значения данных недействительны. Механизм проверки данных работает параллельно привязкам данных; никогда не обрабатывайте данные форм, не проверив, что данные формы в целом действительны (см. раздел «Проверка данных для всей формы»).

Вывод сообщений проверки данных на уровне полей

Применение цветов в обратной связи проверки данных сообщает пользователю, что что-то пошло не так, но ничего не говорит о том, что нужно сделать пользователю. Директива `ngModel` открывает доступ к статусу проверки элементов, к которым она применяется, и может использоваться для отображения рекомендаций при возникновении проблем. Листинг 14.17 добавляет сообщения проверки данных для каждого атрибута, применяемого к элементу `input`, с использованием средств, предоставляемых директивой `ngModel`.

Листинг 14.17. Добавление сообщений проверки данных в файле template.html

```
<style>
    input.ng-dirty.ng-invalid { border: 2px solid #ff0000 }
    input.ng-dirty.ng-valid { border: 2px solid #6bc502 }
</style>

<div class="bg-info p-a-1 m-b-1">Model Data: {{jsonProduct}}</div>

<form novalidate (ngSubmit)="addProduct(newProduct)">
    <div class="form-group">
        <label>Name</label>
        <input class="form-control"
            name="name"
            [(ngModel)]="newProduct.name"
            #name="ngModel"
            required
            minlength="5"
            pattern="^[A-Za-z ]+$"/>
            <ul class="text-danger list-unstyled" *ngIf="name.dirty && name.invalid">
                <li *ngIf="name.errors.required">
                    You must enter a product name
                </li>
                <li *ngIf="name.errors.pattern">
                    Product names can only contain letters and spaces
                </li>
                <li *ngIf="name.errors.minlength">
                    Product names must be at least
                    {{name.errors.minlength.requiredLength}} characters
                </li>
            </ul>
    </div>
    <button class="btn btn-primary" type="submit">
        Create
    </button>
</form>
```

Чтобы проверка данных заработала, необходимо создать ссылочную переменную шаблона для обращения к состоянию проверки данных в выражениях:

```
...
<input class="form-control" name="name" [(ngModel)]="newProduct.name"
    #name="ngModel" required minlength="5" pattern="^[A-Za-z ]+$"/>
...
```

Я создал ссылочную переменную шаблона с именем `name` и присвоил ей значение `ngModel`. Такое использование значения `ngModel` выглядит довольно странно; эта возможность предоставляется директивой `ngModel` для получения доступа к статусу проверки данных. Ситуация немного прояснится после глав 15 и 16, в которых я расскажу, как создавать нестандартные директивы и как они предоставляют доступ к своей функциональности. А в этой главе достаточно знать, что для вывода сообщений проверки данных необходимо создать ссылочную переменную шаблона и присвоить ей `ngModel` для обращения к данным проверки элемента `input`. Объект, присваиваемый ссылочной переменной шаблона, определяет свойства, описанные в табл. 14.6.

Таблица 14.6. Свойства объекта проверки данных

Имя	Описание
path	Свойство возвращает имя элемента
valid	Свойство возвращает true, если содержимое элемента действительно, или false в противном случае
invalid	Свойство возвращает true, если содержимое элемента недействительно, или false в противном случае
pristine	Свойство возвращает true, если содержимое элемента не изменилось
dirty	Свойство возвращает true, если содержимое элемента изменилось
touched	Свойство возвращает true, если пользователь посетил элемент
untouched	Свойство возвращает true, если пользователь не посетил элемент
errors	Свойство возвращает объект, свойства которого соответствуют каждому атрибуту, для которого обнаружена ошибка проверки данных
value	Свойство возвращает значение value элемента, которое используется при определении нестандартных правил проверки данных (см. раздел «Нестандартные классы проверки данных»)

В листинге 14.17 сообщения проверки данных выводятся в списке. Список должен быть видим только в том случае, если при проверке обнаружена хотя бы одна ошибка, поэтому я применяю директиву `ngIf` к элементу `ul` с выражением, использующим свойства `dirty` и `invalid`:

```
...
<ul class="text-danger list-unstyled" *ngIf="name.dirty && name.invalid">
...
```

Внутри элемента `ul` для каждой ошибки проверки данных находится элемент `li`. Каждый элемент `li` содержит директиву `ngIf`, которая использует свойство `errors` из табл. 14.6 следующим образом:

```
...
<li *ngIf="name.errors.required">You must enter a product name</li>
...
```

Свойство `errors.required` определяется только в том случае, если содержимое элемента не прошло проверку `required`; таким образом, видимость элемента `li` связывается с исходом этой конкретной проверки.

СВОЙСТВА БЕЗОПАСНОЙ НАВИГАЦИИ

Свойство errors создается только при наличии ошибок проверки данных; вот почему я проверяю значение свойства invalid в выражении элемента ul. Альтернативное решение основано на использовании свойства безопасной навигации, которое применяется в шаблонах для перехода по цепочке свойств без генерирования ошибок в том случае, если одно из них вернет null. Ниже приведен альтернативный вариант определения шаблона в листинге 14.17, который вместо проверки свойства valid использует свойство безопасной навигации:

```
...
<ul class="text-danger list-unstyled" *ngIf="name.dirty">
    <li *ngIf="name.errors?.required">
        You must enter a product name
    </li>
    <li *ngIf="name.errors?.pattern">
        Product names can only contain letters and spaces
    </li>
    <li *ngIf="name.errors?.minlength">
        Product names must be at least
        {{name.errors.minlength.requiredLength}} characters
    </li>
</ul>
...
```

Присоединение суффикса ? к имени свойства сообщает Angular, что если свойство равно null или undefined, последующие свойства или методы обрабатывать не следует. В данном примере символ ? указывается после свойства errors; это означает, что Angular не будет обращаться к свойствам required, pattern или minlength, если свойство error не определено.

Каждое свойство, определяемое объектом `errors`, возвращает объект. Свойства этого объекта предоставляют подробную информацию о том, почему содержимое не прошло проверку по своему атрибуту. С этой информацией вы сможете сделать сообщения проверки данных более полезными для пользователя. В табл. 14.7 описаны свойства ошибок для каждого атрибута.

Напрямую использовать эти свойства при выводе сообщений для пользователя не удастся — вряд ли пользователь поймет сообщение об ошибке, включающее

Таблица 14.7. Свойства описания ошибок проверки данных для форм Angular

Имя	Описание
required	Свойство возвращает true, если к элементу input был применен атрибут required. Особой пользы от него нет — этот вывод можно сделать из того факта, что свойство required существует
minlength.requiredLength	Свойство возвращает количество символов, необходимых для выполнения требования атрибута minlength
minlength.actualLength	Свойство возвращает количество символов, введенных пользователем
pattern.requiredPattern	Свойство возвращает регулярное выражение, которое было задано для атрибута pattern
pattern.actualValue	Свойство возвращает содержимое элемента

регулярное выражение, хотя такие сообщения пригодятся во время разработки для диагностики ошибок проверки данных. Исключение составляет свойство `minlength.requiredLength`, которое помогает избежать дублирования значения, присвоенного атрибуту `minlength` элемента:

```
...
<li *ngIf="name.errors.minlength">
  Product names must be at least {{name.errors.minlength.requiredLength}}
characters
</li>
...
```

В результате формируется набор сообщений проверки данных, которые выводятся сразу же после того, как пользователь начнет редактировать элемент `input`, и изменяются при каждом нажатии клавиши, как показано на рис. 14.10.

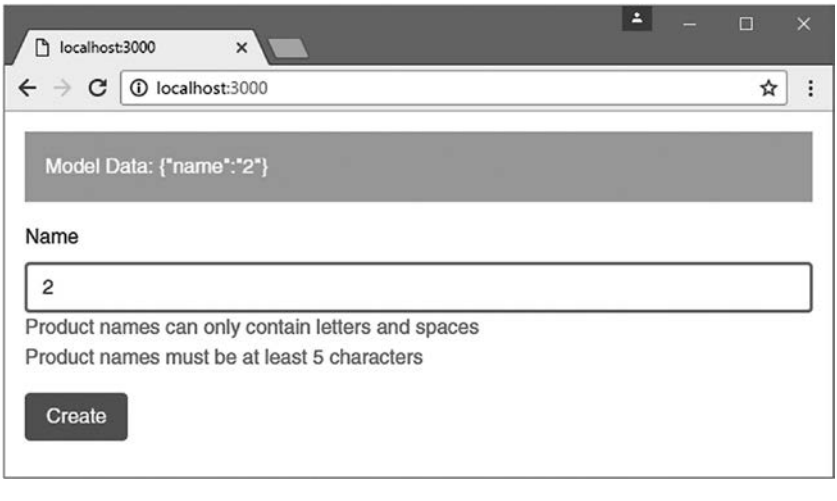


Рис. 14.10. Вывод сообщений проверки данных

Использование компонента для вывода сообщений проверки данных

Включение отдельных элементов для всех возможных ошибок проверки данных приводит к быстрому загромождению сложных форм. Лучше добавить в компонент метод с логикой подготовки сообщений, которые затем выводятся для пользователя при помощи директивы `ngFor` в шаблоне. В листинге 14.18 продемонстрировано добавление метода компонента, который получает информацию состояния проверки для элемента `input` и строит массив сообщений.

Листинг 14.18. Генерирование сообщений об ошибке в файле component.ts

```
import { ApplicationRef, Component } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";

@Component({
    selector: "app",
    templateUrl: "app/template.html"
})
export class ProductComponent {
    model: Model = new Model();

    getProduct(key: number): Product {
        return this.model.getProduct(key);
    }

    getProducts(): Product[] {
        return this.model.getProducts();
    }

    newProduct: Product = new Product();

    get jsonProduct() {
        return JSON.stringify(this.newProduct);
    }

    addProduct(p: Product) {
        console.log("New Product: " + this.jsonProduct);
    }

    getValidationMessages(state: any, thingName?: string) {
        let thing: string = state.path || thingName;
        let messages: string[] = [];
        if (state.errors) {
            for (let errorName in state.errors) {
                switch (errorName) {
                    case "required":
                        messages.push(`You must enter a ${thing}`);
                        break;
                    case "minlength":
                        messages.push(`A ${thing} must be at least
                            ${state.errors['minlength'].requiredLength}
                            characters`);
```

```
                    break;
                case "pattern":
                    messages.push(`The ${thing} contains
                        illegal characters`);
                    break;
            }
        }
    }
    return messages;
  }
}
```

При помощи свойств, описанных в табл. 14.6, метод `getValidationMessages` строит сообщения для всех ошибок и возвращает их в виде строкового массива. Чтобы код получился по возможности универсальным, метод получает значение, описывающее элемент данных, который элемент `input` должен получить от пользователя. Это сообщение используется для генерирования сообщений об ошибках:

```
...
messages.push(`You must enter a ${thing}`);
...
```

В этом примере используется строковая интерполяция JavaScript/ES6: разработчик определяет строковый шаблон без применения оператора `+` для включения данных. Шаблонные строки являются частью стандарта JavaScript/ES6, но компилятор TypeScript преобразует их в традиционные строки JavaScript, чтобы они работали в старых браузерах. Обратите внимание: строка шаблона заключается в обратные апострофы (символ ` вместо обычного символа JavaScript ').

Если при вызове метода аргумент не передается, то метод `getValidationMessages` по умолчанию использует свойство `path` в качестве строкового описания:

```
...
let thing: string = state.path || thingName;
...
```

В листинге 14.19 показано, как использовать `getValidationMessages` в шаблоне для генерирования сообщений проверки данных без определения раздельных элементов и привязок для каждого случая.

Листинг 14.19. Получение сообщений проверки данных от компонента в файле template.html

```
<style>
    input.ng-dirty.ng-invalid { border: 2px solid #ff0000 }
    input.ng-dirty.ng-valid { border: 2px solid #6bc502 }
</style>

<div class="bg-info p-a-1 m-b-1">Model Data: {{jsonProduct}}</div>

<form novalidate (ngSubmit)="addProduct(newProduct)">
    <div class="form-group">
        <label>Name</label>
```

```
        <input class="form-control" [(ngModel)]="newProduct.name"
            name="name"
            #name="ngModel"
            required
            minlength="5"
            pattern="^[A-Za-z ]+$"/>
        <ul class="text-danger list-unstyled" *ngIf="name.dirty && name.invalid">
            <li *ngFor="let error of getValidationMessages(name)">
                {{error}}
            </li>
        </ul>
    </div>
    <button class="btn btn-primary" type="submit">
        Create
    </button>
</form>
```

Внешне ничего не изменилось, но этот метод может использоваться для получения проверочных сообщений для разных элементов. В результате шаблон упрощается, а с чтением и сопровождением кода будет меньше проблем.

Проверка данных для всей формы

Сообщения об ошибках проверки данных для отдельных полей полезны, потому что они помогают понять, где нужно исправить проблему. С другой стороны, они также могут пригодиться для проверки всей формы. Постарайтесь не перегружать пользователя многочисленными сообщениями об ошибках, пока он не попытается отправить форму; в этот момент можно вывести сводку всех обнаруженных проблем. Для этого в листинге 14.20 в компонент добавляются два новых метода.

Листинг 14.20. Расширение компонента в файле component.ts

```
import { ApplicationRef, Component } from "@angular/core";
import { NgForm } from "@angular/forms";
import { Model } from "./repository.model";
import { Product } from "./product.model";

@Component({
    selector: "app",
    templateUrl: "app/template.html"
})
export class ProductComponent {
    model: Model = new Model();

    // ...Другие методы опущены для краткости...

    formSubmitted: boolean = false;

    submitForm(form: NgForm) {
        this.formSubmitted = true;
        if (form.valid) {
            this.addProduct(this.newProduct);
            this.newProduct = new Product();
```

```
            form.reset();
            this.formSubmitted = false;
        }
    }
}
```

Свойство `formSubmitted` указывает, была ли отправлена форма, и используется для предотвращения проверки всей формы до того, как пользователь попытается отправить данные.

Метод `submitForm` будет вызываться при отправке формы пользователем. В аргументе он получает объект `NgForm`. Этот объект представляет форму и определяет набор свойств проверки данных, которые используются для описания общего статуса проверки данных формы; например, свойство `invalid` равно `true`, если в каких-либо элементах, содержащихся в форме, присутствуют ошибки проверки данных. Кроме свойства `validation`, `NgForm` предоставляет метод `reset`, который сбрасывает статус проверки данных формы и возвращает ее к исходному «чистому» состоянию.

В результате вся форма будет проверяться при отправке данных пользователем. При отсутствии ошибок проверки новый объект добавляется в модель данных перед сбросом формы, чтобы его можно было использовать повторно.

В листинге 14.21 показаны изменения, которые необходимо внести в шаблон для использования этих новых возможностей и реализации проверки на уровне формы.

Листинг 14.21. Выполнение проверки данных на уровне формы в файле template.html

```
<style>
    input.ng-dirty.ng-invalid { border: 2px solid #ff0000 }
    input.ng-dirty.ng-valid { border: 2px solid #6bc502 }
</style>

<form novalidate #form="ngForm" (ngSubmit)="submitForm(form)">

    <div class="bg-danger p-a-1 m-b-1"
        *ngIf="formSubmitted && form.invalid">
        There are problems with the form
    </div>

    <div class="form-group">
        <label>Name</label>
        <input class="form-control"
            name="name"
            [(ngModel)]="newProduct.name"
            #name="ngModel"
            required
            minlength="5"
            pattern="^[A-Za-z ]+$"/>
        <ul class="text-danger list-unstyled"
                *ngIf="(formSubmitted || name.dirty) && name.invalid">
            <li *ngFor="let error of getValidationMessages(name)">
                {{error}}
```

```
            </li>
        </ul>
    </div>
    <button class="btn btn-primary" type="submit">
        Create
    </button>
</form>
```

Элемент `form` теперь определяет ссылочную переменную с именем `form`, которой присваивается `ngForm`. Так директива `ngForm` предоставляет доступ к своей функциональности — через механизм, который будет описан в главе 15. А пока важно запомнить, что через ссылочную переменную `form` можно получить доступ к информации проверки данных всей формы.

В листинге также изменяется выражение привязки `ngSubmit`: в нем вызывается метод `submitForm`, определяемый контроллером, с передачей переменной в шаблон:

```
...
<form novalidate ngForm="productForm" #form="ngForm" (ngSubmit)="submitForm(form)">
...
```

Этот объект, передаваемый в аргументе метода `submitForm`, используется для обращения к статусу проверки данных формы, а также для сброса формы для ее последующего использования.

В листинге 14.21 также добавляется элемент `div`, использующий свойство `formSubmitted` из компонента наряду со свойством `valid` (которое предоставляется переменной шаблона `form`) для вывода предупреждения при обнаружении недействительных данных на форме — но только после ее отправки.

Кроме того, привязка `ngIf` была обновлена для вывода сообщений уровня отдельных полей, чтобы эти сообщения отображались при отправке данных формы, даже если сам элемент при этом не редактировался. В результате формируется сводка проверки данных, которая отображается только при отправке формы с недействительными данными, как показано на рис. 14.11.

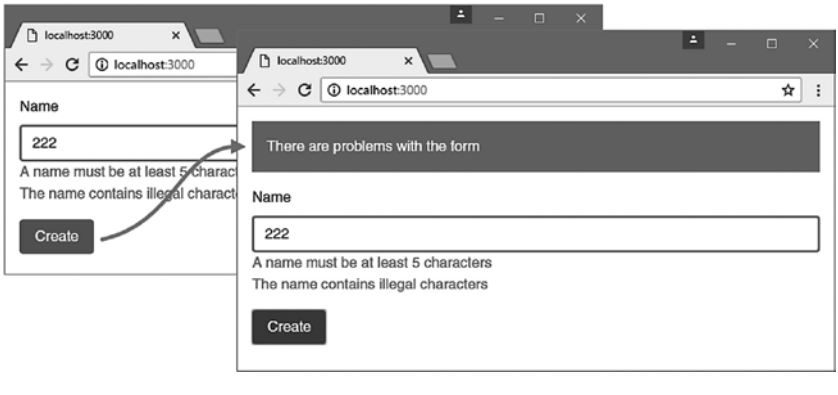


Рис. 14.11. Вывод сводки проверки данных

Вывод сводки проверки данных

На сложных формах бывает полезно вывести сводку всех ошибок проверки данных, на которые необходимо отреагировать.

Объект `NgForm`, присвоенный ссылочной переменной шаблона, открывает доступ к отдельным элементам через свойство с именем `controls`. Свойство возвращает объект со свойствами для каждого отдельного элемента формы. Например, свойство `name` в данном примере представляет элемент `input`; этому свойству присваивается объект, который представляет элемент и определяет те же свойства проверки данных, которые доступны для отдельных элементов. В листинге 14.22 в компонент добавляется метод, который получает объект, присвоенный ссылочным переменным шаблонов элемента `form`, и использует свое свойство `controls` для генерирования списка сообщений об ошибках для всей формы.

Листинг 14.22. Генерирование сообщений проверки данных уровня формы в файле component.ts

```
import { ApplicationRef, Component } from "@angular/core";
import { NgForm } from "@angular/forms";
import { Model } from "./repository.model";
import { Product } from "./product.model";

@Component({
    selector: "app",
    templateUrl: "app/template.html"
})
export class ProductComponent {
    model: Model = new Model();

    // ...Другие методы опущены для краткости...

    getFormValidationMessages(form: NgForm): string[] {
        let messages: string[] = [];
        Object.keys(form.controls).forEach(k => {
            this.getValidationMessages(form.controls[k], k)
                .forEach(m => messages.push(m));
        });
        return messages;
    }
}
```

Метод `getFormValidationMessages` строит список сообщений, вызывая метод `getValidationMessages` из листинга 14.18 для каждого элемента формы. Метод `Object.keys` создает массив на базе свойств, определяемых объектом, возвращаемым свойством `controls`, перебранным методом `forEach`.

ПРИМЕЧАНИЕ

Мне приходится передавать методу getValidationMessages имя, которое должно использоваться в сообщениях, потому что свойство path не задается для объектов, получаемых из свойства controls.

В листинге 14.23 этот метод используется для включения отдельных сообщений в верхней части формы; эти сообщения становятся видимыми после того, как пользователь нажмет кнопку Create.

Листинг 14.23. Отображение сообщений проверки данных уровня формы в файле template.html

```
<style>
    input.ng-dirty.ng-invalid { border: 2px solid #ff0000 }
    input.ng-dirty.ng-valid { border: 2px solid #6bc502 }
</style>

<form novalidate #form="ngForm" (ngSubmit)="submitForm(form)">

    <div class="bg-danger p-a-1 m-b-1"
        *ngIf="formSubmitted && form.invalid">
        There are problems with the form
        <ul>
            <li *ngFor="let error of getFormValidationMessages(form)">
                {{error}}
            </li>
        </ul>
    </div>

    <div class="form-group">
        <label>Name</label>
        <input class="form-control"
            name="name"
            [(ngModel)]="newProduct.name"
            #name="ngModel"
            required
            minlength="5"
            pattern="^[A-Za-z ]+$"/>
        <ul class="text-danger list-unstyled"
                *ngIf="(formSubmitted || name.dirty) && name.invalid">
            <li *ngFor="let error of getValidationMessages(name)">
                {{error}}
            </li>
        </ul>
    </div>
    <button class="btn btn-primary" type="submit">
        Create
    </button>
</form>
```

В результате сообщения проверки данных отображаются наряду с элементом `input` и собираются воедино в верхней части формы после отправки данных формы (рис. 14.12).

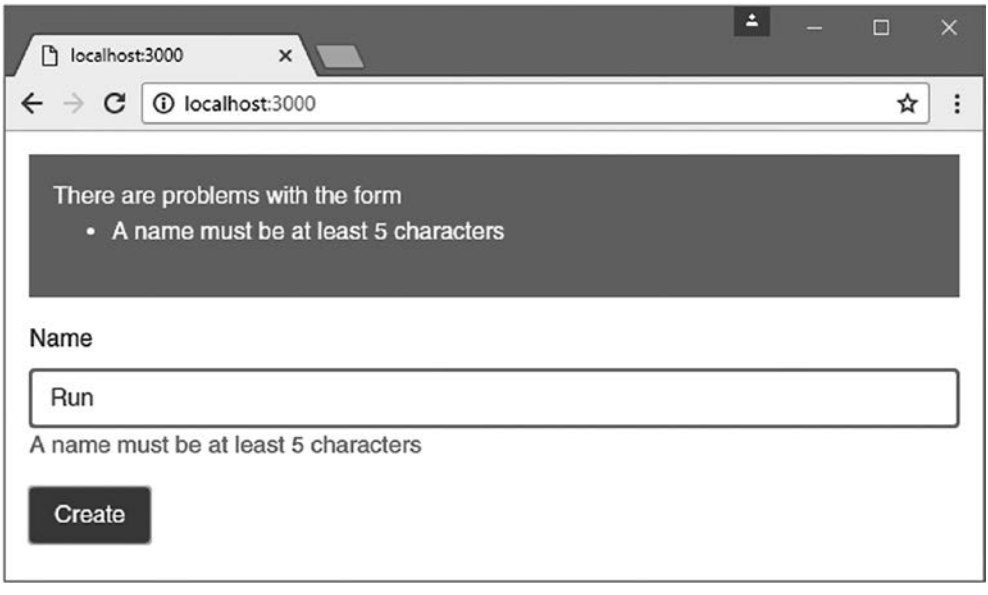


Рис. 14.12. Отображение сводки проверки данных

Блокировка кнопки отправки данных

В этом разделе остается внести последнее изменение: заблокировать кнопку после того, как пользователь отправил данные формы, чтобы он не мог нажать ее повторно до устранения всех ошибок. Этот прием часто применяется на практике; он не оказывает особого влияния на работу приложения, которое все равно не примет данные с формы с недействительными значениями, но ясно дает понять пользователю, что продолжение работы возможно только после того, как будут устранены все проблемы.

В листинге 14.24 к элементу `button` применяется привязка свойства, а также добавляется элемент `input` для свойства `price` для демонстрации того, как этот метод масштабируется с ростом количества элементов на форме.

Листинг 14.24. Блокировка кнопки и добавление элемента input в файле template.html

```
<style>
    input.ng-dirty.ng-invalid { border: 2px solid #ff0000 }
    input.ng-dirty.ng-valid { border: 2px solid #6bc502 }
</style>
```

```
<form novalidate #form="ngForm" (ngSubmit)="submitForm(form)">

    <div class="bg-danger p-a-1 m-b-1"
        *ngIf="formSubmitted && form.invalid">
        There are problems with the form
        <ul>
            <li *ngFor="let error of getFormValidationMessages(form)">
                {{error}}
            </li>
        </ul>
    </div>

    <div class="form-group">
        <label>Name</label>
        <input class="form-control"
            name="name"
            [(ngModel)]="newProduct.name"
            #name="ngModel"
            required
            minlength="5"
            pattern="^[A-Za-z ]+$"/>
        <ul class="text-danger list-unstyled"
                *ngIf="(formSubmitted || name.dirty) && name.invalid">
            <li *ngFor="let error of getValidationMessages(name)">
                {{error}}
            </li>
        </ul>
    </div>

    <div class="form-group">
        <label>Price</label>
        <input class="form-control" name="price" [(ngModel)]="newProduct.price"
            #price="ngModel" required pattern="^[0-9\.]+$"/>
        <ul class="text-danger list-unstyled"
                *ngIf="(formSubmitted || price.dirty) && price.invalid">
            <li *ngFor="let error of getValidationMessages(price)">
                {{error}}
            </li>
        </ul>
    </div>

    <button class="btn btn-primary" type="submit"
        [disabled]="formSubmitted && form.invalid"
        [class.btn-secondary]="formSubmitted && form.invalid">
            Create
    </button>
</form>
```

Для большей выразительности я использовал привязку класса для назначения класса `btn-secondary` элементу `button` в том случае, если данные были отправле-

ны, а форма содержит недействительные данные. Результат применения стиля Bootstrap показан на рис. 14.13.

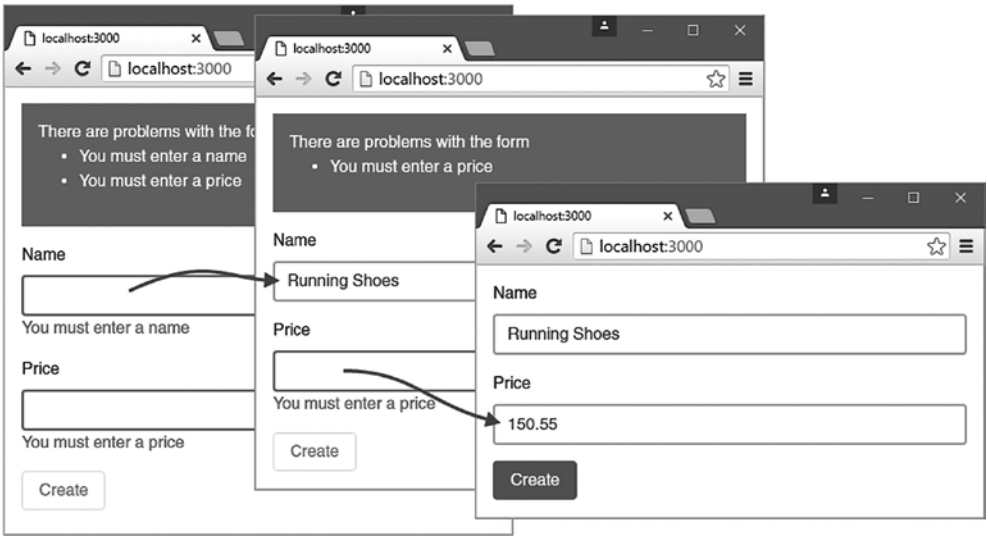


Рис. 14.13. Блокировка кнопки отправки данных

Использование форм на базе моделей

Форма из предыдущего раздела строилась из атрибутов и элементов HTML, которые определяли поля с информацией и использовались для применения ограничений проверки данных. К преимуществам такого решения относится его простота и логичность. Недостаток заключается в том, что большие формы становятся сложными и создают проблемы с сопровождением: каждому полю требуется собственный блок контента для управления его макетом и критериями проверки, а также для вывода сообщений проверки.

Angular также предоставляет другое решение — *модели на базе форм*. Подробности форм и критериев проверки определяются в коде, а не в шаблоне. Такое решение лучше масштабируется, но требует некоторой работы на подготовительной стадии, а результаты не так естественны, как при определении всей информации в шаблоне. В последующих разделах мы создадим и применим модель, которая описывает форму и необходимую проверку данных.

Включение поддержки форм на базе моделей

Поддержка форм на базе моделей требует объявления новой зависимости в модуле Angular приложения (листинг 14.25).

Листинг 14.25. Включение поддержки форм на базе моделей в файле app.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { ProductComponent } from "./component";
import { FormsModule, ReactiveFormsModule } from "@angular/forms";

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

Функциональность форм на базе моделей определяется в модуле `ReactiveFormsModule`, который определяется в модуле JavaScript `@angular/forms`, добавленном в проект в начале главы.

Определение классов модели для формы

Начнем с определения классов, описывающих форму, чтобы шаблон оставался по возможности простым. Вам не обязательно следовать этому решению во всех подробностях, но если вы собираетесь применить форму на базе модели, как можно большую часть обработки следует выполнять в модели и свести сложность шаблона к минимуму. Я добавил файл form.model.ts в папку app и добавил код, приведенный в листинге 14.26.

Листинг 14.26. Содержимое файла form.model.ts в папке app

```
import { FormControl, FormGroup, Validators } from "@angular/forms";

export class ProductFormControl extends FormControl {
    label: string;
    modelProperty: string;

    constructor(label:string, property:string, value: any, validator: any) {
        super(value, validator);
        this.label = label;
        this.modelProperty = property;
    }
}

export class ProductFormGroup extends FormGroup {

    constructor() {
        super({
            name: new ProductFormControl("Name", "name", "", Validators.required),

            category: new ProductFormControl("Category", "category", "",
                Validators.compose([Validators.required,
                    Validators.pattern("^[A-Za-z ]+$"),
                    Validators.minLength(3),
```

```
                    Validators.maxLength(10)])),

            price: new ProductFormControl("Price", "price", "",
                Validators.compose([Validators.required,
                    Validators.pattern("^[0-9\.]+$")]))
        });
    }

    get productControls(): ProductFormControl[] {
        return Object.keys(this.controls)
            .map(k => this.controls[k] as ProductFormControl);
    }
}
```

Два класса, определенные в листинге, расширяют классы, используемые во внутренних механизмах Angular для управления формами и их контентом. Класс `FormControl` представляет один элемент формы (например, элемент `input`), а класс `FormGroup` используется для управления элементом формы и его содержимым.

Новые субклассы добавляют функциональность, которая упрощает генерирование формы HTML на программном уровне. Класс `ProductFormControl` расширяет класс `FormControl` свойствами, которые задают текст элемента `label`, связанного с элементом `input`, и имя свойства класса `Product`, представляемого элементом `input`. Класс `ProductControlGroup` расширяет `FormGroup` свойством, представляющим массив объектов `ProductFormControl`, которые были определены в форме и которые будут использоваться в шаблоне для генерирования контента с использованием директивы `ngFor`.

Важной частью этого класса является конструктор класса `ProductFormGroup`, отвечающий за создание модели, которая будет использоваться для создания и проверки формы. Конструктор класса `FormGroup`, являющегося суперклассом `ProductFormGroup`, получает объект, имена свойств которого соответствуют именам элементов `input` в шаблоне; каждому из них присваивается объект `ProductFormControl`, который представляет данный элемент и определяет необходимые проверки данных. Первое свойство в объекте, передаваемом конструктору суперкласса, самое простое:

```
...
name: new ProductFormControl("Name", "name", "", Validators.required),
...
```

Свойству присвоено имя `name`; оно сообщает Angular, что свойство соответствует элементу `input` с именем `name` в шаблоне. Аргументы конструктора `ProductFormControl` задают содержимое элемента `label`, связываемого с элементом `input` (`Name`), имя свойства класса `Product`, к которому привязывается элемент `input` (`name`), исходное значение привязки данных (пустая строка) и необходимые проверки данных. Angular определяет в модуле `@angular/forms` класс `Validators`, который содержит свойства для всех встроенных проверок данных; эти свойства перечислены в табл. 14.8.

Таблица 14.8. Свойства объекта Validator

Имя	Описание
Validators.required	Свойство соответствует атрибуту required и проверяет, что пользователь ввел обязательное значение
Validators.minLength	Свойство соответствует атрибуту minlength и проверяет минимальное количество символов
Validators.maxLength	Свойство соответствует атрибуту maxlength и проверяет максимальное количество символов
Validators.pattern	Свойство соответствует атрибуту pattern и проверяет наличие совпадения регулярного выражения в строке

Объекты `Validators` могут объединяться методом `Validators.compose` для выполнения нескольких проверок с одним элементом:

```
...
category: new ProductFormControl("Category", "category", "",
    Validators.compose([Validators.required,
        Validators.pattern("^[A-Za-z ]+$"),
        Validators.minLength(3),
        Validators.maxLength(10)])),
...
```

Метод `Validators.compose` получает массив объектов `Validators`. Аргументы конструктора, определяемые значениями `pattern`, `minLength` и `maxLength`, соответствуют значениям атрибутов. В итоге для данного элемента это означает, что значение является обязательным, должно содержать только алфавитные символы и пробелы и содержит от 3 до 10 символов.

На следующем шаге методы, генерирующие сообщения об ошибках проверки, перемещаются из компонента в новые классы модели формы (листинг 14.27). Это делается для того, чтобы весь код, относящийся к форме, находился в одном месте, а компонент был по возможности простым. (Я также добавил поддержку сообщения проверки для `maxLength` в метод `getValidationMessages` класса `ProductFormControl`.)

Листинг 14.27. Перемещение методов сообщений в файл form.model.ts

```
import { FormControl, FormGroup, Validators } from "@angular/forms";

export class ProductFormControl extends FormControl {
    label: string;
    modelProperty: string;

    constructor(label:string, property:string, value: any, validator: any) {
        super(value, validator);
        this.label = label;
        this.modelProperty = property;
    }

    getValidationMessages() {
        let messages: string[] = [];
        if (this.errors) {
```

```
            for (let errorName in this.errors) {
                switch (errorName) {
                    case "required":
                        messages.push(`You must enter a ${this.label}`);
                        break;
                    case "minlength":
                        messages.push(`A ${this.label} must be at least
                            ${this.errors['minlength'].requiredLength}
                            characters`);
                        break;
                    case "maxlength":
                        messages.push(`A ${this.label} must be no more than
                            ${this.errors['maxlength'].requiredLength}
                            characters`);
                        break;
                    case "pattern":
                        messages.push(`The ${this.label} contains
                             illegal characters`);
                        break;
                }
            }
        }
        return messages;
    }
}

export class ProductFormGroup extends FormGroup {
    constructor() {
        super({
            name: new ProductFormControl("Name", "name", "", Validators.required),

            category: new ProductFormControl("Category", "category", "",
                Validators.compose([Validators.required,
                    Validators.pattern("^[A-Za-z ]+$"),
                    Validators.minLength(3),
                    Validators.maxLength(10)])),

            price: new ProductFormControl("Price", "price", "",
                Validators.compose([Validators.required,
                    Validators.pattern("^[0-9\.]+$")]))
        });
    }

    get productControls(): ProductFormControl[] {
        return Object.keys(this.controls)
            .map(k => this.controls[k] as ProductFormControl);
    }

    getFormValidationMessages(form: any) : string[] {
        let messages: string[] = [];
        this.productControls.forEach(c => c.getValidationMessages()
            .forEach(m => messages.push(m)));
        return messages;
    }
}
```

Сообщения генерируются почти так же, как прежде; незначительные изменения отражают тот факт, что код теперь является частью модели формы, а не компонента.

Использование модели для проверки

Итак, модель формы готова, и мы можем использовать ее для проверки формы. Листинг 14.28 показывает, как обновить класс компонента для использования форм на базе модели и как открыть доступ к классам модели формы из шаблона. При этом удаляются методы, генерирующие сообщения об ошибках проверки данных; они перемещаются в классы модели формы из листинга 14.27.

Листинг 14.28. Использование модели формы в файле component.ts

```
import { ApplicationRef, Component } from "@angular/core";
import { NgForm } from "@angular/forms";
import { Model } from "./repository.model";
import { Product } from "./product.model";
import { ProductFormGroup } from "./form.model";

@Component({
    selector: "app",
    templateUrl: "app/template.html"
})
export class ProductComponent {
    model: Model = new Model();
    form: ProductFormGroup = new ProductFormGroup();

    getProduct(key: number): Product {
        return this.model.getProduct(key);
    }

    getProducts(): Product[] {
        return this.model.getProducts();
    }

    newProduct: Product = new Product();

    get jsonProduct() {
        return JSON.stringify(this.newProduct);
    }

    addProduct(p: Product) {
        console.log("New Product: " + this.jsonProduct);
    }

    formSubmitted: boolean = false;

    submitForm(form: NgForm) {
        this.formSubmitted = true;
        if (form.valid) {
            this.addProduct(this.newProduct);
            this.newProduct = new Product();
            form.reset();
```

```
            this.formSubmitted = false;
        }
    }
}
```

Класс `ProductFormGroup` импортируется из модуля `form.model` и используется для определения свойства с именем `form`, что позволяет использовать класс модели из шаблона.

В листинге 14.29 шаблон обновляется для использования функциональности модели для проверки данных, с заменой конфигурации проверки на базе атрибутов, определенной в шаблоне.

Листинг 14.29. Использование модели формы в файле template.html

```
<style>
    input.ng-dirty.ng-invalid { border: 2px solid #ff0000 }
    input.ng-dirty.ng-valid { border: 2px solid #6bc502 }
</style>

<form novalidate [formGroup]="form" (ngSubmit)="submitForm(form)">

    <div class="bg-danger p-a-1 m-b-1" *ngIf="formSubmitted && form.invalid">
        There are problems with the form
        <ul>
            <li *ngFor="let error of form.getFormValidationMessages()">
                {{error}}
            </li>
        </ul>
    </div>

    <div class="form-group">
        <label>Name</label>
        <input class="form-control" name="name" [(ngModel)]="newProduct.name"
            formControlName="name" />
        <ul class="text-danger list-unstyled"
                *ngIf="(formSubmitted || form.controls['name'].dirty) &&
                     form.controls['name'].invalid">
            <li *ngFor="let error of form.controls['name'].getValidationMessages()">
                {{error}}
            </li>
        </ul>
    </div>

    <div class="form-group">
        <label>Category</label>
        <input class="form-control" name="name" [(ngModel)]="newProduct.category"
            formControlName="category" />
        <ul class="text-danger list-unstyled"
                *ngIf="(formSubmitted || form.controls['category'].dirty) &&
                     form.controls['category'].invalid">
            <li *ngFor="let error of form.controls['category']
                    .getValidationMessages()">
```

```
                {{error}}
            </li>
        </ul>
    </div>

    <div class="form-group">
        <label>Price</label>
        <input class="form-control" name="price" [(ngModel)]="newProduct.price"
            formControlName="price" />
        <ul class="text-danger list-unstyled"
                *ngIf="(formSubmitted || form.controls['price'].dirty) &&
                 form.controls['price'].invalid">
            <li *ngFor="let error of form.controls['price'].
getValidationMessages()">
                {{error}}
            </li>
        </ul>
    </div>

    <button class="btn btn-primary" type="submit"
        [disabled]="formSubmitted && form.invalid"
        [class.btn-secondary]="formSubmitted && form.invalid">
            Create
    </button>
</form>
```

Первые изменения вносятся в элемент `form`. Для проверки на базе модели необходимо присутствие директивы `formGroup`:

```
...
<form novalidate [formGroup]="form" (ngSubmit)="submitForm(form)">
...
```

Значение, присвоенное директиве `formGroup`, представляет собой свойство `form` компонента; оно возвращает объект `ProductFormGroup`, поставляющий информацию проверки данных для формы.

Следующие изменения относятся к элементам `input`. Отдельные атрибуты проверки данных и переменная шаблона, которой присваивалось специальное значение `ngForm`, были удалены. Также добавляется новый атрибут `forControlName`, который идентифицирует элемент `input` в системе формы на базе модели по имени, использованном в `ProductFormGroup` в листинге 14.26.

```
...
<input class="form-control" name="name" [(ngModel)]="newProduct.name"
    formControlName="name" />
...
```

Этот атрибут также позволяет Angular добавлять и удалять классы проверки данных для элемента `input`. В данном случае атрибуту `formControlName` присваивается имя `name`, которое сообщает Angular, что элемент должен проверяться с использованием указанных объектов.

```
...
name: new ProductFormControl("Name", "name", "", Validators.required),
...
```

Свойство `form` компонента предоставляет доступ к данным проверки для каждого элемента:

```
...
<li *ngFor="let error of form.controls['name'].getValidationMessages()">
{{error}} </li>
...
```

Класс `FormGroup` предоставляет свойство `controls`, которое возвращает коллекцию объектов `FormControl`, находящихся под его управлением (с индексированием по имени). Отдельные объекты `FormControl` извлекаются из коллекции и либо анализируются для получения информации о состоянии проверки, либо используются для генерирования проверочных сообщений.

Как часть изменений в листинге 14.29, я добавил все три элемента `input`, необходимые для получения данных для создания новых объектов `Product`; каждый элемент проверяется по модели проверки данных (рис. 14.14).

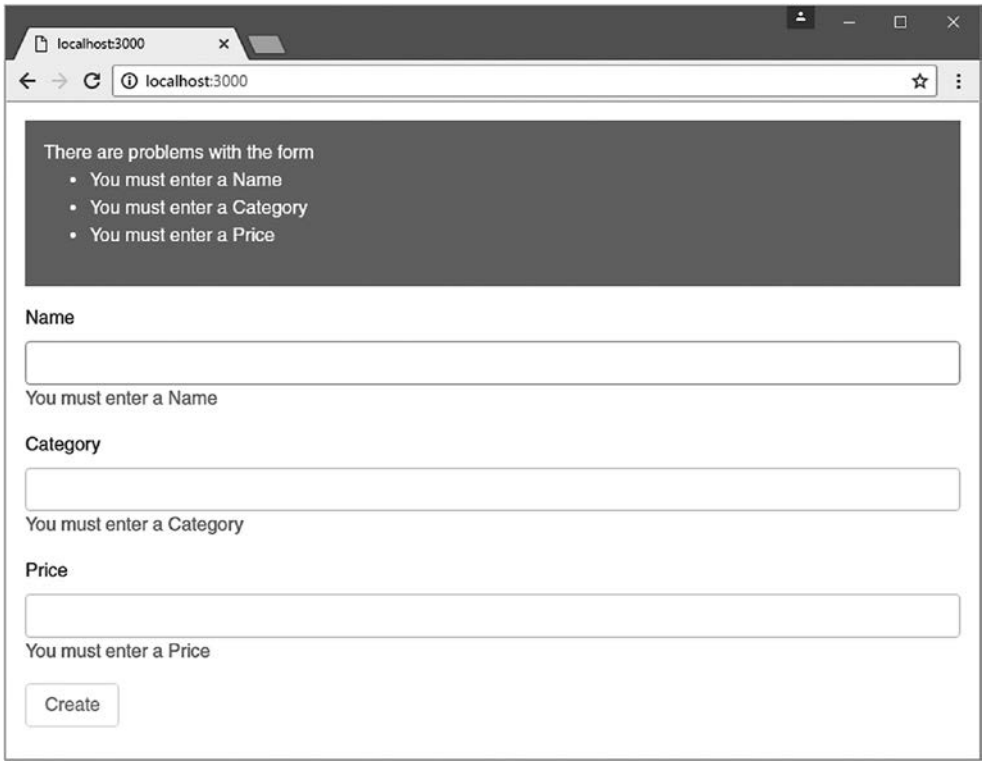


Рис. 14.14. Проверка данных с использованием формы

Генерирование элементов по модели

Листинг 14.29 содержит много повторений. Атрибуты проверки были вынесены в код, но каждый элемент `input` по-прежнему требует вспомогательной инфраструктуры контента для управления макетом и вывода проверочных сообщений для пользователя.

Следующим шагом должно стать упрощение шаблона: модель формы будет использоваться для генерирования элементов формы, а не только для их проверки. В листинге 14.30 показано, как объединить стандартные директивы Angular с моделью формы для генерирования формы на программном уровне.

Листинг 14.30. Использование модели для генерирования формы в файле template.html

```
<style>
    input.ng-dirty.ng-invalid { border: 2px solid #ff0000 }
    input.ng-dirty.ng-valid { border: 2px solid #6bc502 }
</style>

<form novalidate [formGroup]="form" (ngSubmit)="submitForm(form)">

    <div class="bg-danger p-a-1 m-b-1" *ngIf="formSubmitted && form.invalid">
        There are problems with the form
        <ul>
            <li *ngFor="let error of form.getFormValidationMessages()">
                {{error}}
            </li>
        </ul>
    </div>

    <div class="form-group" *ngFor="let control of form.productControls">
        <label>{{control.label}}</label>
        <input class="form-control"
            [(ngModel)]="newProduct[control.modelProperty]"
            name="{{control.modelProperty}}"
            formControlName="{{control.modelProperty}}" />
        <ul class="text-danger list-unstyled"
                *ngIf="(formSubmitted || control.dirty) && control.invalid">
            <li *ngFor="let error of control.getValidationMessages()">
                {{error}}
            </li>
        </ul>
    </div>

    <button class="btn btn-primary" type="submit"
        [disabled]="formSubmitted && form.invalid"
        [class.btn-secondary]="formSubmitted && form.invalid">
            Create
    </button>
</form>
```

В этом листинге директива ngFor используется для создания элементов формы по описанию, предоставленному классами модели ProductFormControl и ProductFormGroup. Каждый элемент настраивается с теми же атрибутами, что и в листинге 14.29, но значения берутся из описаний модели. Это позволяет упростить шаблон и использовать модель как для определения элементов формы, так и для их проверки.

После того как базовая модель формы будет готова, ее можно расширять в соответствии с потребностями приложения. Например, в нее можно добавлять новые элементы; расширять субкласс FormControl дополнительной информацией (скажем, значениями атрибута type элементов input); генерировать элементы select для некоторых полей; предоставлять подсказки, которые помогут пользователю понять назначение элементов, и т. д.

Нестандартные правила проверки данных

Angular поддерживает возможность создания нестандартных правил, с помощью которых можно сформировать политику проверки данных, адаптированную к целям конкретного приложения (в отличие от универсальных правил проверки, предоставляемых встроенными средствами). Для демонстрации создайте файл с именем limit.formvalidator.ts в папке app и включите в него определение класса из листинга 14.31.

Листинг 14.31. Содержимое файла limit.formvalidator.ts в папке app

```
import { FormControl } from "@angular/forms";

export class LimitValidator {

    static Limit(limit:number) {
        return (control:FormControl) : {[key: string]: any} => {
            let val = Number(control.value);
            if (val != NaN && val > limit) {
                return {"limit": {"limit": limit, "actualValue": val}};
            } else {
                return null;
            }
        }
    }
}
```

Нестандартные классы проверки данных

Нестандартные классы проверки создают функции, используемые для выполнения проверки данных. В данном случае класс LimitValidator определяет статический метод-фабрику Limit, который возвращает функцию проверки. Аргументом метода Limit является наибольшее значение, которое должно проходить проверку.

При вызове функции проверки, возвращаемой методом `Limit`, Angular передает в аргументе метод `FormControl`. Нестандартная функция проверки в листинге использует свойство `value` для получения значения, введенного пользователем, преобразования его в число и сравнения с допустимым предельным значением.

Функции проверки возвращают `null` для действительных значений или объект с информацией об ошибке для недействительных значений. Чтобы описать ошибку проверки данных, объект определяет свойство, которое указывает, какое правило было нарушено (`limit` в данном случае), и присваивает ему другой объект с расширенной информацией.

Свойство `limit` возвращает объект; у этого объекта имеется свойство `limit`, которому присваивается предельное значение, и свойство `actualValue`, которому присваивается значение, введенное пользователем.

Применение нестандартной проверки данных

В листинге 14.32 продемонстрировано расширение модели формы для поддержки нового класса проверки данных и его применение к элементу `input` для свойства `price`.

Листинг 14.32. Применение нестандартного класса проверки данных в файле form.model.ts

```
import { FormControl, FormGroup, Validators } from "@angular/forms";
import { LimitValidator } from "./limit.formvalidator";

export class ProductFormControl extends FormControl {
    label: string;
    modelProperty: string;

    constructor(label:string, property:string, value: any, validator: any) {
        super(value, validator);
        this.label = label;
        this.modelProperty = property;
    }

    getValidationMessages() {
        let messages: string[] = [];
        if (this.errors) {
            for (let errorName in this.errors) {
                switch (errorName) {
                    case "required":
                        messages.push(`You must enter a ${this.label}`);
                        break;
                    case "minlength":
                        messages.push(`A ${this.label} must be at least
                            ${this.errors['minlength'].requiredLength}
                            characters`);
                        break;
```

```
                    case "maxlength":
                        messages.push(`A ${this.label} must be no more than
                            ${this.errors['maxlength'].requiredLength}
                            characters`);
                        break;
                    case "limit":
                        messages.push(`A ${this.label} cannot be more
                                than ${this.errors['limit'].limit}`);
                        break;
                    case "pattern":
                        messages.push(`The ${this.label} contains
                             illegal characters`);
                        break;
                }
            }
        }
        return messages;
    }
}

export class ProductFormGroup extends FormGroup {
    constructor() {
        super({
            name: new ProductFormControl("Name", "name", "", Validators.required),
            category: new ProductFormControl("Category", "category", "",
                Validators.compose([Validators.required,
                    Validators.pattern("^[A-Za-z ]+$"),
                    Validators.minLength(3),
                    Validators.maxLength(10)])),
            price: new ProductFormControl("Price", "price", "",
                Validators.compose([Validators.required,
                    LimitValidator.Limit(100),
                    Validators.pattern("^[0-9\.]+$")]))
        });
    }

    get productControls(): ProductFormControl[] {
        return Object.keys(this.controls)
            .map(k => this.controls[k] as ProductFormControl);
    }

    getFormValidationMessages(form: any) : string[] {
        let messages: string[] = [];
        this.productControls.forEach(c => c.getValidationMessages()
            .forEach(m => messages.push(m)));
        return messages;
    }
}
```

В результате значение, введенное в поле `Price`, ограничивается предельным значением 100, а при больших значениях выводится сообщение об ошибке проверки данных, показанное на рис. 14.15.

Рис. 14.15. Нестандартное сообщение проверки

Итоги

В этой главе я рассказал о том, как Angular обеспечивает взаимодействие с пользователем с использованием событий и форм. Вы узнали, как создавать привязки событий и двусторонние привязки и как они могут упрощаться при помощи директивы `ngModel`. Также была рассмотрена поддержка управления формами HTML и проверки данных в Angular. В следующей главе вы научитесь создавать нестандартные директивы.

15 Создание директив атрибутов

В этой главе я опишу, как нестандартные директивы используются для дополнения функциональности, предоставляемой встроенными директивами Angular. В этой главе основное внимание уделяется *директивам атрибутов* — простейшей разновидности директив, предназначенной для изменения внешнего вида или поведения одного элемента. В главе 16 вы узнаете, как создавать *структурные директивы*, предназначенные для изменения макета документа HTML.

Компоненты также являются разновидностью директив; вы узнаете, как они работают, в главе 17.

В этих главах для описания работы нестандартных директив мы воспроизведем функциональность, предоставляемую некоторыми встроенными директивами. В реальных проектах вы вряд ли станете этим заниматься, но такое упражнение дает удобный ориентир для объяснения процесса. В табл. 15.1 директивы атрибутов представлены в контексте.

Таблица 15.1. Директивы атрибутов в контексте

Вопрос	Ответ
Что это такое?	Директивы атрибутов — классы, способные изменять поведение или внешний вид элемента, к которому они применяются. Привязки стилей и классов, описанные в главе 12, являются примерами директив атрибутов
Для чего они нужны?	Встроенные директивы реализуют большинство стандартных задач, встречающихся в разработке веб-приложений, но они не исчерпывают всех возможностей. Нестандартные директивы позволяют определять возможности, специализированные для конкретного приложения
Как они используются?	Директивы атрибутов представляют собой классы, к которым был применен декоратор @Directive. Они включаются в свойстве directives компонента, отвечающего за шаблон, и применяются с использованием селектора CSS
Есть ли у них недостатки или скрытые проблемы?	Основная проблема при создании нестандартных директив — искушение написать код для решения задач, которые более эффективно решаются с помощью таких функций, как входные/выходные свойства или привязки управляющих элементов

Таблица 15.1 (*окончание*)

Вопрос	Ответ
Есть ли альтернативы?	Angular поддерживает еще два типа директив (структурные директивы и директивы компонентов), которые могут лучше подойти для конкретных задач. Иногда можно добиться желаемого эффекта при помощи комбинации встроенных директив, если вы предпочитаете обойтись без написания специализированного кода (хотя результат может быть ненадежным, а полученная разметка HTML окажется неудобочитаемой и трудной в сопровождении)

В табл. 15.2 приведена краткая сводка материала главы.

Таблица 15.2. Сводка материала главы

Проблема	Решение	Листинг
Создание директивы атрибута	Примените к классу декоратор @Directive	1–5
Обращение к значениям атрибутов управляющих элементов	Примените декоратор @Attribute к параметру конструктора	6–9
Создание входного свойства с привязкой к данным	Примените декоратор @Input к свойству класса	10–11
Получение уведомлений об изменении значения входного свойства, привязанного к данным	Реализуйте метод ngOnChanges	12
Определение события	Примените декоратор @Output	13, 14
Создание привязки свойства или привязки события для управляющего элемента	Примените декоратор @HostBinding или @HostListener	15–19
Экспортирование функциональности директивы для использования в шаблоне	Используйте свойство exportAs декоратора @Directive	20, 21

Подготовка проекта

Как и во всех предшествующих главах, мы продолжим использовать проект example из предыдущей главы. Чтобы подготовиться к этой главе, добавьте в шаблон таблицу для вывода товаров из модели данных и удалите сообщения проверки данных уровня формы (листинг 15.1).

Листинг 15.1. Подготовка шаблона в файле template.html

```
<style>
    input.ng-dirty.ng-invalid { border: 2px solid #ff0000 }
    input.ng-dirty.ng-valid { border: 2px solid #6bc502 }
</style>

<div class="col-xs-6">
```

```
    <form novalidate [formGroup]="form" (ngSubmit)="submitForm(form)">
        <div class="form-group" *ngFor="let control of form.productControls">
            <label>{{control.label}}</label>
            <input class="form-control"
                [(ngModel)]="newProduct[control.modelProperty]"
                name="{{control.modelProperty}}"
                formControlName="{{control.modelProperty}}" />
            <ul class="text-danger list-unstyled"
                    *ngIf="(formSubmitted || control.dirty) && control.invalid">
                <li *ngFor="let error of control.getValidationMessages()">
                    {{error}}
                </li>
            </ul>
        </div>
        <button class="btn btn-primary" type="submit"
            [disabled]="formSubmitted && !form.valid"
            [class.btn-secondary]="formSubmitted && form.invalid">
                Create
        </button>
    </form>
</div>

<div class="col-xs-6">
    <table class="table table-sm table-bordered table-striped">
        <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
        <tr *ngFor="let item of getProducts(); let i = index">
            <td>{{i + 1}}</td>
            <td>{{item.name}}</td>
            <td>{{item.category}}</td>
            <td>{{item.price}}</td>
        </tr>
    </table>
</div>
```

В листинге используется макет сетки Bootstrap для размещения формы и таблицы рядом друг с другом. В листинге 15.2 удаляется свойство `jsonProperty`, а метод `addProduct` изменяется и в модель данных добавляется новый объект.

Листинг 15.2. Изменение модели данных в файле component.ts

```
import { ApplicationRef, Component } from "@angular/core";
import { NgForm } from "@angular/forms";
import { Model } from "./repository.model";
import { Product } from "./product.model";
import { ProductFormGroup } from "./form.model";

@Component({
    selector: "app",
    templateUrl: "app/template.html"
})

export class ProductComponent {
    model: Model = new Model();
    form: ProductFormGroup = new ProductFormGroup();

    getProduct(key: number): Product {
```

```
        return this.model.getProduct(key);
    }

    getProducts(): Product[] {
        return this.model.getProducts();
    }

    newProduct: Product = new Product();
    addProduct(p: Product) {
        this.model.saveProduct(p);
    }

    formSubmitted: boolean = false;

    submitForm(form: NgForm) {
        this.formSubmitted = true;
        if (form.valid) {
            this.addProduct(this.newProduct);
            this.newProduct = new Product();
            form.reset();
            this.formSubmitted = false;
        }
    }
}
```

Чтобы запустить приложение, перейдите в папку проекта и выполните следующую команду:

```
npm start
```

Запускается веб-сервер для разработки, и открывается новое окно браузера с формой, показанной на рис. 15.1. При отправке данных формы выполняется проверка, после которой либо выводятся сообщения об ошибках, либо в модель данных добавляется новый объект, который отображается в таблице.

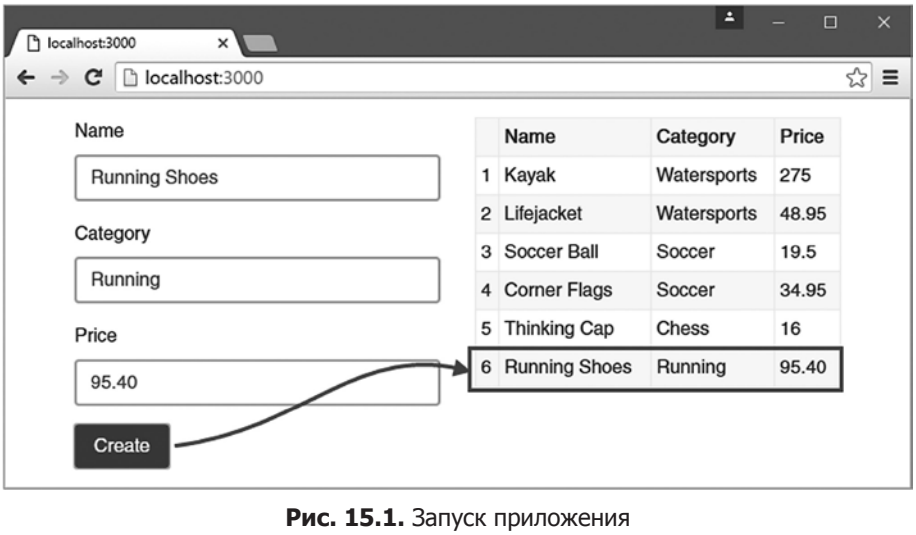


Рис. 15.1. Запуск приложения

Создание простой директивы атрибута

Проще всего с ходу взяться за дело: создать директиву и посмотреть, как она работает. Создайте файл attr.directive.ts в папке app folder и включите в него код из листинга 15.3. Имя файла указывает на то, что файл содержит директиву. Первая часть имени файла — attr — сообщает, что перед вами пример директивы атрибута.

Листинг 15.3. Содержимое файла attr.directive.ts в папке app

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

@Directive({
    selector: "[pa-attr]",
})
export class PaAttrDirective {

    constructor(element: ElementRef) {
        element.nativeElement.classList.add("bg-success");
    }
}
```

Директивы представляют собой классы, к которым был применен декоратор `@Directive`. У декоратора имеется обязательное свойство `selector`, которое определяет способ применения директивы к элементам, заданный стандартным селектором стиля CSS. Я использую селектор `[pa-attr]`, который выбирает любой элемент, обладающий атрибутом с именем `pa-attr`, независимо от типа элемента или значения, присвоенного атрибуту.

Нестандартным директивам назначается префикс, по которому их можно легко узнать. Префикс может быть любым — подходит любое значение, осмысленное для вашего приложения. Я выбрал для своей директивы префикс `Pa` (от названия этой книги на английском языке); этот префикс используется в атрибуте, заданном свойством декоратора `selector`, и в имени класса атрибута. Регистр префикса изменяется в зависимости от использования: в имени атрибута используется начальный символ нижнего регистра (`pa-attr`), а в имени класса директивы — начальный символ верхнего регистра (`PaAttrDirective`).

ПРИМЕЧАНИЕ

Префикс Ng/ng зарезервирован для встроенной функциональности Angular и не может использоваться в приложениях.

Конструктор директивы определяет один параметр `ElementRef`, который предоставляется Angular при создании нового экземпляра директивы и который представляет управляющий элемент.

Класс `ElementRef` определяет одно свойство `nativeElement`, которое возвращает объект, используемый браузером для представления элемента в модели DOM. Этот объект предоставляет доступ к методам и свойствам, которые манипулируют

элементом и его содержимым; к их числу принадлежит свойство `classList`, которое используется для управления принадлежностью элемента к классам:

```
...
element.nativeElement.classList.add("bg-success");
...
```

Итак, класс `PaAttrDirective` представляет собой директиву, которая применяется к элементам, имеющим атрибут `pa-attr`, и добавляет эти элементы к классу `bg-success`. Последний класс используется библиотекой Bootstrap для назначения цвета фона элементам.

Применение нестандартной директивы

Применение нестандартной директивы состоит из двух шагов. Первый шаг — внесение изменений в шаблон, чтобы в нем появились один и более элементов с селектором, используемым директивой. Для директивы из нашего примера это означает добавление в элемент атрибута `pa-attr` (листинг 15.4).

Листинг 15.4. Добавление атрибута для директивы в файл template.html

```
...
<table class="table table-sm table-bordered table-striped">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
    <tr *ngFor="let item of getProducts(); let i = index" pa-attr>
        <td>{{i + 1}}</td>
        <td>{{item.name}}</td>
        <td>{{item.category}}</td>
        <td>{{item.price}}</td>
    </tr>
</table>
...
```

Селектор директивы подходит для любого элемента с атрибутом независимо от того, присвоено ли ему значение (и какое именно).

Второй шаг — применение директивы для изменения конфигурации модуля Angular (листинг 15.5).

Листинг 15.5. Настройка компонента в файле app.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { ProductComponent } from "./component";
import { FormsModule, ReactiveFormsModule  } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";

@NgModule({
    imports: [BrowserModule, FormsModule, ReactiveFormsModule],
    declarations: [ProductComponent, PaAttrDirective],
    bootstrap: [ProductComponent]
})
export class AppModule { }
```

Свойство `declarations` декоратора `NgModule` объявляет директивы и компоненты, используемые приложением. Не беспокойтесь, если сходство и различие между этими директивами и компонентами еще не совсем понятно; в главе 17 все прояснится.

После того как оба этих шага будут выполнены, атрибут `pa-attr`, применяемый к элементу `tr` в шаблоне, инициирует выполнение нестандартной директивы, которая использует DOM API для добавления элемента в класс `bg-success`. Так как элемент `tr` является частью микрошаблона, используемого директивой `ngFor`, результат затрагивает все строки таблицы (рис. 15.2).

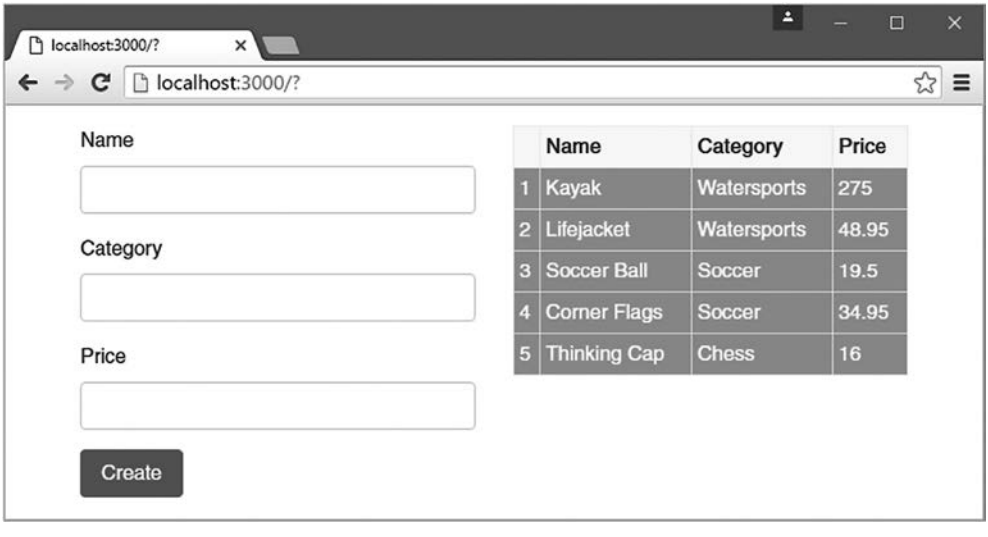


Рис. 15.2. Применение нестандартной директивы

Обращение к данным приложения в директиве

Пример из предыдущего раздела демонстрирует базовую структуру директивы, но он не делает ничего, что нельзя было бы сделать простым применением привязки свойства класса к элементу `tr`. Чтобы директива приносила пользу, она должна взаимодействовать с управляющим элементом и остальным кодом приложения.

Чтение атрибутов управляющего элемента

Самый простой способ реального использования директивы — ее настройка с учетом значений атрибутов, примененных к управляющему элементу; в результате каждый элемент директивы получает свои данные конфигурации и адаптирует свое поведение соответствующим образом.

В листинге 15.6 директива применяется к некоторым элементам `td` в таблице шаблона; также добавляется атрибут, который задает класс, к которому должен добавляться управляющий элемент. Селектор директивы означает, что совпадение будет найдено для любого элемента, обладающего атрибутом `pa-attr`, независимо от типа тега. Таким образом, директива будет работать как с элементами `td`, так и с элементами `tr`.

Листинг 15.6. Добавление атрибутов в файл template.html

```
...
<table class="table table-sm table-bordered table-striped">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
    <tr *ngFor="let item of getProducts(); let i = index" pa-attr>
        <td>{{i + 1}}</td>
        <td>{{item.name}}</td>
        <td pa-attr pa-attr-class="bg-warning">{{item.category}}</td>
        <td pa-attr pa-attr-class="bg-info">{{item.price}}</td>
    </tr>
</table>
...
```

Атрибут `pa-attr` был применен к двум элементам `td` вместе с новым атрибутом `pa-attr-class`, задающим класс, который директива должна назначить управляющему элементу.

В листинге 15.7 показаны изменения, которые необходимо внести в директиву для получения значения атрибута `pa-attr-class` и его использования для изменения элемента.

Листинг 15.7. Чтение атрибута в файле attr.directive.ts

```
import { Directive, ElementRef, Attribute } from "@angular/core";

@Directive({
    selector: "[pa-attr]",
})
export class PaAttrDirective {

    constructor(element: ElementRef, @Attribute("pa-attr-class") bgClass: string) {
        element.nativeElement.classList.add(bgClass || "bg-success");
    }
}
```

Чтобы получить значение атрибута `pa-attr-class`, я добавил новый параметр конструктора с именем `bgClass`, к которому применяется декоратор `@Attribute`. Декоратор определяется в модуле `@angular/core` и задает имя атрибута, который должен предоставить значение параметра конструктора при создании нового экземпляра класса директивы. Angular создает новый экземпляр декоратора для каждого элемента, который подходит по селектору, и использует атрибуты этого элемента для получения значений аргументов конструктора директивы, помеченных декоратором `@Attribute`.

В конструкторе значение атрибута передается методу `classList.add` со значением по умолчанию, которое позволяет применять директиву к элементу с атрибутом `pa-attr`, но не с атрибутом `pa-attr-class`.

В результате класс, в который включаются элементы, теперь может задаваться атрибутом. Результат показан на рис. 15.3.

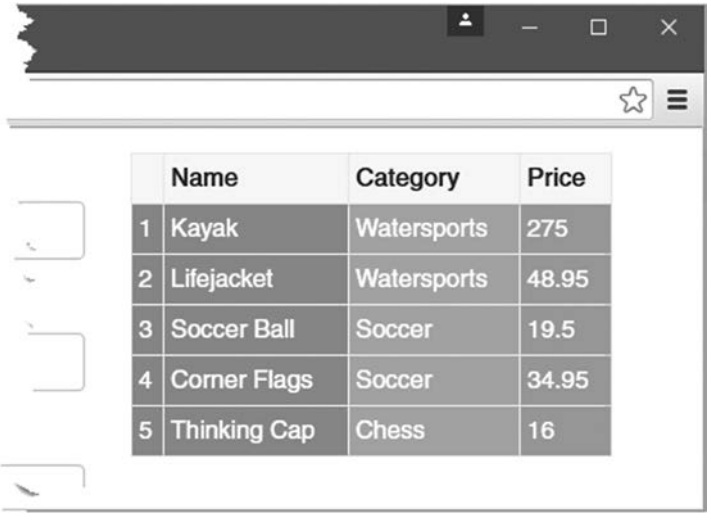


Рис. 15.3. Настройка директивы с использованием атрибута управляющего элемента

Использование одного атрибута управляющего элемента

Использовать два разных атрибута для применения директивы и ее настройки неэффективно. Разумнее возложить обе обязанности на один атрибут, как показано в листинге 15.8.

Листинг 15.8. Повторное использование атрибута в файле attr.directive.ts

```
import { Directive, ElementRef, Attribute } from "@angular/core";

@Directive({
    selector: "[pa-attr]",
})
export class PaAttrDirective {

    constructor(element: ElementRef, @Attribute("pa-attr") bgClass: string) {
        element.nativeElement.classList.add(bgClass || "bg-success");
    }
}
```

Декоратор `@Attribute` теперь задает атрибут `pa-attr` источником значения параметра `bgClass`. В листинге 15.9 в шаблон вносятся изменения в соответствии с новой двойной ролью атрибута.

Листинг 15.9. Применение директивы в файле template.html

```
...
<table class="table table-sm table-bordered table-striped">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
    <tr *ngFor="let item of getProducts(); let i = index" pa-attr>
        <td>{{i + 1}}</td>
        <td>{{item.name}}</td>
        <td pa-attr="bg-warning">{{item.category}}</td>
        <td pa-attr="bg-info">{{item.price}}</td>
    </tr>
</table>
...
```

Внешне в результате этого примера ничего не изменилось, однако применение директивы в шаблоне HTML упростилось.

Создание входных свойств с привязкой к данным

Главное ограничение чтения атрибутов с использованием `@Attribute` — статичность значений. Истинная сила директив Angular проявляется в поддержке выражений, которые обновляются в соответствии с изменениями в состоянии приложения и могут реагировать на них изменением управляющего элемента.

Директивы получают выражения через *входные свойства с привязкой к данным*, также называемые просто *входными свойствами*. В листинге 15.10 шаблон приложения изменяется таким образом, что атрибуты `pa-attr`, примененные к элементам `tr` и `td`, содержат выражения вместо статических имен классов.

Листинг 15.10. Использование выражений в файле template.html

```
...
<table class="table table-sm table-bordered table-striped">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
    <tr *ngFor="let item of getProducts(); let i = index"
        [pa-attr]="getProducts().length < 6 ? 'bg-success' : 'bg-warning'">
        <td>{{i + 1}}</td>
        <td>{{item.name}}</td>
        <td [pa-attr]="item.category == 'Soccer' ? 'bg-info' : null">
            {{item.category}}
        </td>
        <td [pa-attr]="'bg-info'">{{item.price}}</td>
    </tr>
</table>
...
```

В этом листинге встречаются три выражения. Первое, примененное к элементу `tr`, использует количество объектов, возвращенных методом `getProducts` компонента, для выбора класса.

```
...
<tr *ngFor="let item of getProducts(); let i = index"
    [pa-attr]="getProducts().length < 6 ? 'bg-success' : 'bg-warning'">
...
```

Второе выражение, которое применяется к элементу `td` для столбца Category, назначает класс `bg-info` объектам `Product`, у которых свойство `Category` возвращает `Soccer`, и `null` для всех остальных значений.

```
...
<td [pa-attr]="item.category == 'Soccer' ? 'bg-info' : null">
...
```

Третье, и последнее, выражение возвращает фиксированное строковое значение, которое заключается в апострофы, потому что оно представляет выражение, а не статическое значение атрибута.

```
...
<td [pa-attr]="'bg-info'">{{item.price}}</td>
...
```

Обратите внимание: имя атрибута заключается в квадратные скобки. Это объясняется тем, что для получения выражения в директиве применяется привязка данных (как и у встроенных директив, описанных в главах 13 и 14).

ПРИМЕЧАНИЕ

Разработчики нередко забывают о квадратных скобках. Без них Angular просто передает необработанный текст выражения директиве без его вычисления. Это первое, что следует проверить, если вы столкнулись с ошибкой при применении нестандартной директивы.

Чтобы реализовать другую сторону привязки данных, необходимо создать входное свойство в классе директивы и объяснить Angular, как управлять его значением (листинг 15.11).

Листинг 15.11. Определение входного свойства в файле attr.directive.ts

```
import { Directive, ElementRef, Attribute, Input } from "@angular/core";

@Directive({
    selector: "[pa-attr]"
})
export class PaAttrDirective {

    constructor(private element: ElementRef) {}

    @Input("pa-attr")
    bgClass: string;

    ngOnInit() {
        this.element.nativeElement.classList.add(this.bgClass || "bg-success");
    }
}
```

Входные свойства определяются применением декоратора `@Input` к свойству и использованием его для задания имени атрибута, содержащего выражение. В ли-

стинге определяется одно входное свойство, которое приказывает Angular задать свойству `bgClass` директивы значение выражения, содержащегося в атрибуте `pa-attr`.

ПРИМЕЧАНИЕ

Передавать аргумент декоратору @Input не обязательно, если имя свойства соответствует имени атрибута управляющего элемента. Таким образом, если вы примените @Input() к свойству с именем myVal, Angular будет искать у управляющего элемента атрибут myVal.

Роль конструктора в этом примере изменилась. Когда Angular создает новый экземпляр класса директивы, конструктор вызывается для создания нового объекта директивы; только после этого будет задано значение входного свойства. Это означает, что конструктор не может обращаться к значению входного свойства, потому что его значение не будет задано до завершения конструктора и создания нового объекта директивы. Для решения этой проблемы директивы могут реализовать *методы жизненного цикла*, которые используются Angular для передачи директивам полезной информации после их создания и во время работы приложения. Эти методы перечислены в табл. 15.3.

Таблица 15.3. Методы жизненного цикла директив

Имя	Описание
ngOnInit	Метод вызывается после того, как Angular задаст исходные значения всех входных свойств, объявленных директивой
ngOnChanges	Метод вызывается при изменении значения входного свойства, а также непосредственно перед вызовом метода ngOnInit
ngDoCheck	Метод вызывается, когда Angular запускает свой процесс обнаружения изменений, чтобы у директив была возможность обновить любое состояние, не связанное напрямую со входным свойством
ngAfterContentInit	Метод вызывается после инициализации контента директивы. Пример использования приведен в главе 16, раздел «Получение уведомлений об изменении запросов»
ngAfterContentChecked	Метод вызывается после анализа контента директивы в процессе обнаружения изменений
ngOnDestroy	Метод вызывается непосредственно перед тем, как Angular уничтожает директиву

Чтобы назначить класс управляющего элемента, директива в листинге 15.11 реализует метод `ngOnInit`, который вызывается после того, как Angular задаст значение свойства `bgClass`. Конструктор все равно должен получать объект `ElementRef` для получения доступа к управляющему элементу, который присваивается свойству с именем `element`.

В результате Angular создает объект директивы для каждого элемента `tr`, вычисляет выражения, заданные атрибутом `pa-attr`, использует результаты для задания

значения входных свойств, а затем вызывает методы `ngOnInit`, чтобы директивы могли отреагировать на новые значения входных свойств.

Чтобы понаблюдать за эффектом, воспользуйтесь формой для добавления нового продукта в приложение. Так как изначально модель состоит из пяти объектов, выражение для элемента `tr` выберет класс `bg-success`. При добавлении нового объекта Angular создает другой экземпляр класса директивы и вычисляет выражение для задания значения входного свойства; так как модель теперь состоит из пяти объектов, выражение выберет класс `bg-warning`, который выделит новую строку другим цветом фона (рис. 15.4).

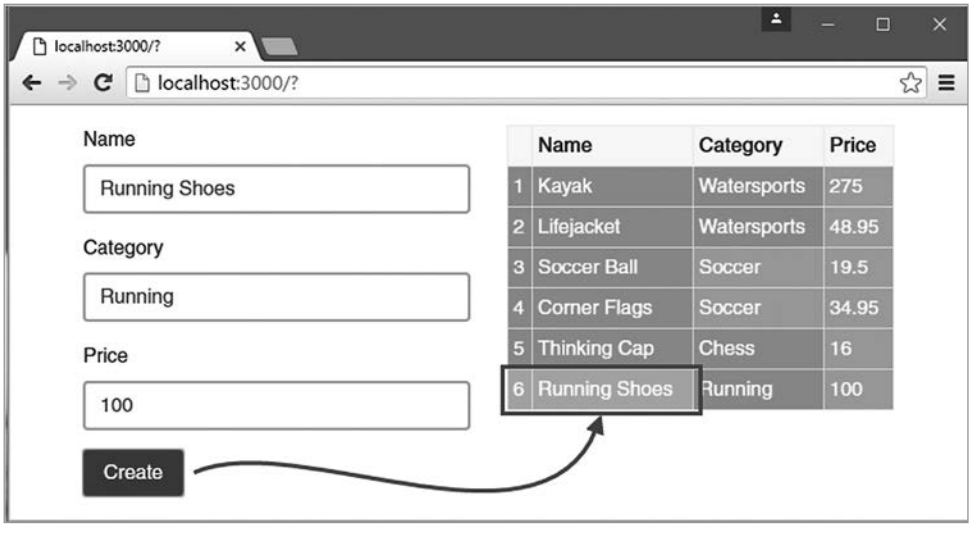


Рис. 15.4. Использование входного свойства в нестандартной директиве

Реакция на изменения входных свойств

В рассмотренном примере происходит нечто странное: добавление нового объекта влияет на внешний вид только новых элементов, но не влияет на уже существующие. Во внутренней реализации Angular обновляет значение свойства `bgClass` для каждой из созданных директив (по одной для каждого элемента `td` в столбце таблицы), но директивы этого не заметили, потому что изменение значения свойства не приводит к автоматической реакции директив.

Для обработки изменений директива должна реализовать метод `ngOnChanges` для получения оповещений об изменении входного свойства (листинг 15.12).

Листинг 15.12. Получение уведомлений об изменениях в файле attr.directive.ts

```
import { Directive, ElementRef, Attribute, Input,
         SimpleChange } from "@angular/core";

@Directive({
    selector: "[pa-attr]"
```

```
})
export class PaAttrDirective {

    constructor(private element: ElementRef) {}

    @Input("pa-attr")
    bgClass: string;

    ngOnChanges(changes: {[property: string]: SimpleChange }) {
        let change = changes["bgClass"];
        let classList = this.element.nativeElement.classList;
        if (!change.isFirstChange() && classList.contains(change.previousValue)) {
            classList.remove(change.previousValue);
        }
        if (!classList.contains(change.currentValue)) {
            classList.add(change.currentValue);
        }
    }
}
```

Метод `ngOnChanges` вызывается однократно перед вызовом `ngOnInit`, а затем каждый раз при обнаружении изменений во входных свойствах любой директивы. В параметре `ngOnChanges` передается объект, имена свойств которого соответствуют каждому изменившемуся входному свойству, а значениями являются объекты `SimpleChange`, определяемые `@angular/core`. В TypeScript эта структура данных представляется так:

```
...
ngOnChanges(changes: {[property: string]: SimpleChange }) {
...
```

Класс `SimpleChange` определяет поля и методы, перечисленные в табл. 15.4.

Таблица 15.4. Свойства и методы класса SimpleChange

Имя	Описание
previousValue	Свойство возвращает предыдущее значение входного свойства
currentValue	Свойство возвращает текущее значение входного свойства
isFirstChange()	Метод возвращает true, если используется вызов метода ngOnChanges, происходящий перед методом ngOnInit

Чтобы понять механизм передачи информации об изменениях методу `ngOnChanges`, проще всего сериализовать объект в формате JSON, а потом проанализировать его.

```
...
{
    "target": {
        "previousValue":"bg-success",
        "currentValue":"bg-warning"
    }
}
...
```

Этот прием исключает метод `isFirstChange`, но он помогает разобраться в том, как свойства объекта-аргумента используются для описания изменений входных свойств.

При обработке изменения значений входных свойств директива обязательно должна отменить эффект предыдущих обновлений. В нашем примере это означает удаление элемента из класса `previousValue` и добавление его в класс `currentValue`.

Важно использовать метод `isFirstChange`, чтобы не отменить значение, которое еще не было применено, так как метод `ngOnChanges` впервые вызывается при первом присваивании значения входному свойству.

Что же дает обработка уведомлений об изменениях? Директива начинает реагировать на повторное вычисление выражения и обновление входных свойств. Теперь при добавлении нового товара в приложение будут обновляться цвета фона всех элементов `tr` (рис. 15.5).

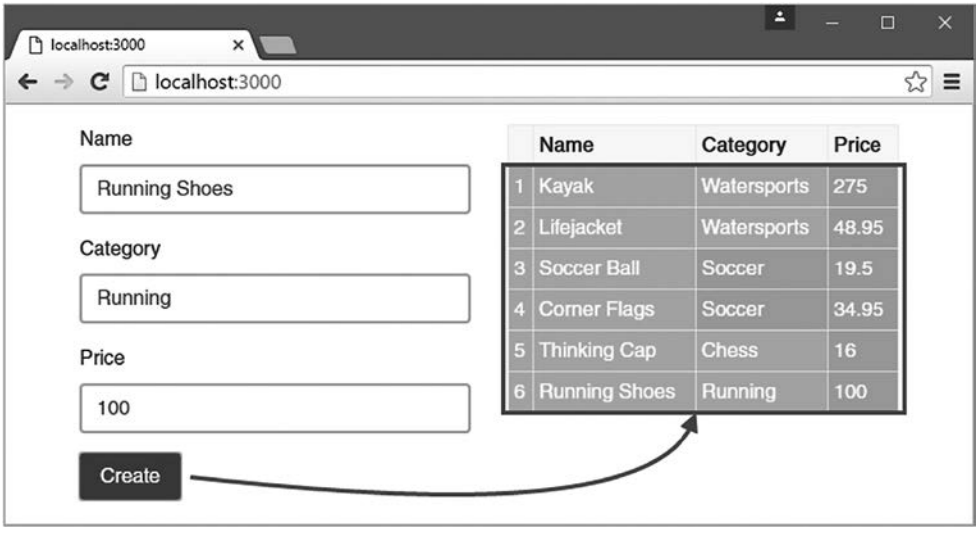


Рис. 15.5. Реакция на изменения входных свойств

Создание нестандартных событий

Механизм *выходных свойств* Angular позволяет директивам добавлять к управляющим элементам нестандартные события, которые могут использоваться для передачи информации о важных изменениях в другие части приложения. Выходные свойства определяются декоратором `@Output`, который определяется в модуле `@angular/core` (листинг 15.13).

Листинг 15.13. Определение выходного свойства в файле attr.directive.ts

```
import { Directive, ElementRef, Attribute, Input,
         SimpleChange, Output, EventEmitter } from "@angular/core";
import { Product } from "./product.model";
```

```
@Directive({
    selector: "[pa-attr]"
})
export class PaAttrDirective {

    constructor(private element: ElementRef) {
        this.element.nativeElement.addEventListener("click", e => {
            if (this.product != null) {
                this.click.emit(this.product.category);
            }
        });
    }

    @Input("pa-attr")
    bgClass: string;

    @Input("pa-product")
    product: Product;

    @Output("pa-category")
    click = new EventEmitter<string>();

    ngOnChanges(changes: {[property: string]: SimpleChange }) {
        let change = changes["bgClass"];
        let classList = this.element.nativeElement.classList;
        if (!change.isFirstChange() && classList.contains(change.previousValue)) {
            classList.remove(change.previousValue);
        }
        if (!classList.contains(change.currentValue)) {
            classList.add(change.currentValue);
        }
    }
}
```

Класс `EventEmitter` предоставляет механизм событий для директив Angular. В листинге создается объект `EventEmitter`, который присваивается переменной с именем `click`:

```
...
@Output("pa-category")
click = new EventEmitter<string>();
...
```

Параметр типа `string` указывает, что слушатели события будут получать строку при инициировании события. Директивы могут передавать своим слушателям событий объекты любого типа, но на практике чаще всего применяются значения `string` и `number`, объекты модели данных и объекты JavaScript `Event`.

Нестандартное событие в листинге инициируется при щелчке кнопкой мыши на управляющем элементе; событие предоставляет своим слушателям категорию объекта `Product`, который использовался для создания строки таблицы директивой `ngFor`. В результате директива реагирует на событие DOM для управляющего

элемента и генерирует свое собственное нестандартное событие в ответ. Слушатель события DOM назначается в конструкторе класса директивы при помощи стандартного метода `addEventListener` браузера:

```
...
constructor(private element: ElementRef) {
    this.element.nativeElement.addEventListener("click", e => {
        if (this.product != null) {
            this.click.emit(this.product.category);
        }
    });
}
...
```

Директива определяет входное свойство для получения объекта `Product`, категория которого будет передаваться в событии. (Директива может обратиться к значению входного свойства в конструкторе, потому что Angular обеспечит инициализацию свойства до вызова функции, назначенной для обработки события DOM.)

Самая важная команда в листинге — та, которая использует объект `EventEmitter` для отправки события методом `EventEmitter.emit` (краткое описание метода приведено в табл. 15.5). Аргумент метода `emit` содержит значение, которое должны получить слушатели события, — в данном случае это значение свойства `category`.

Таблица 15.5. Метод EventEmitter

Имя	Описание
emit(value)	Метод инициирует нестандартное событие, связанное с EventEmitter; слушатели получают объект или значение, переданное в аргументе метода

Объединяя все сказанное, мы получаем декоратор `@Output`, который создает связь между свойством `EventEmitter` класса директивы и именем, которое будет использоваться для привязки события в шаблоне:

```
...
@Output("pa-category")
click = new EventEmitter<string>();
...
```

Аргумент декоратора задает имя атрибута, которое будет использоваться в привязке события, применяемой к управляемому элементу. Аргумент можно опустить, если имя свойства TypeScript совпадает с именем, назначаемым нестандартному событию. В листинге указано имя `pa-category`, что позволяет ссылаться на событие по имени `click` в классе директивы, но использовать более содержательное имя за ее пределами.

Привязка нестандартного события

Angular упрощает привязку к нестандартным событиям, позволяя использовать такой же синтаксис привязки, как для встроенных событий (см. главу 14). В листинге 15.14 в элемент `tr` добавляется атрибут `pa-product`, в котором директиве передается объект `Product`, а также добавляется привязка для события `pa-category`.

Листинг 15.14. Привязка к нестандартному событию в файле template.html

```
...
<table class="table table-sm table-bordered table-striped">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
    <tr *ngFor="let item of getProducts(); let i = index"
            [pa-attr]="getProducts().length < 6 ? 'bg-success' : 'bg-warning'"
            [pa-product]="item" (pa-category)="newProduct.category=$event">
        <td>{{i + 1}}</td>
        <td>{{item.name}}</td>
        <td [pa-attr]="item.category == 'Soccer' ? 'bg-info' : null">
            {{item.category}}
        </td>
        <td [pa-attr]="'bg-info'">{{item.price}}</td>
    </tr>
</table>
...
```

Конструкция `$event` используется для обращения к значению, переданному директивой методу `EventEmitter.emit`. Это означает, что `$event` в данном примере соответствует значению `string`, содержащему категорию товара в данном примере. Значение, полученное от события, назначается свойству `newProduct.category` компонента; это приводит к обновлению привязки данных одного из элементов `input`. Таким образом, при щелчке на строке таблицы на форме выводится категория товара (рис. 15.6).

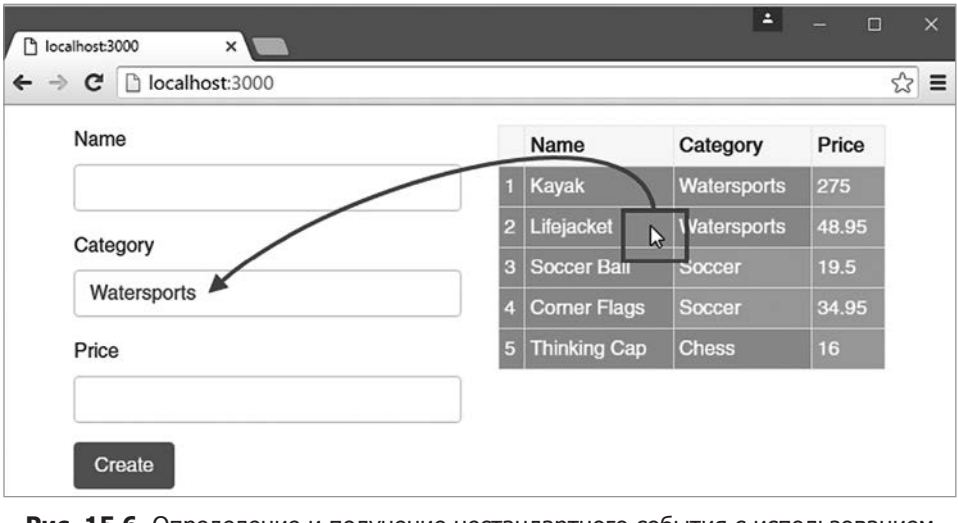


Рис. 15.6. Определение и получение нестандартного события с использованием выходного свойства

Создание привязок управляющих элементов

Директива из нашего примера использует API модели DOM браузера для манипуляций с управляющим элементом — как для добавления и удаления принадлежности к классам, так и для получения события `click`. Работа с DOM API в приложении Angular — полезный прием, но это означает, что директива может использоваться только в приложениях, выполняемых в браузере. Предполагается, что инфраструктура Angular должна работать в широком диапазоне разных исполнительных сред и не все они должны предоставлять DOM API.

Даже если вы уверены, что директива имеет доступ к модели DOM, тех же результатов можно добиться более элегантным способом, основанным на стандартных возможностях директив Angular: привязках свойств и событий. Вместо того чтобы использовать DOM для добавления и удаления классов, можно воспользоваться привязкой класса для управляющего элемента. А вместо использования метода `addEventListener` для обработки щелчков можно воспользоваться привязкой события.

«За кулисами» Angular реализует эти возможности через DOM API, когда директива используется в браузере, — или задействует эквивалентный механизм, когда директива используется в другой среде. Привязки управляющего элемента определяются с использованием двух декораторов, `@HostBinding` и `@HostListener`; оба декоратора определяются в модуле `@angular/core` (листинг 15.15).

Листинг 15.15. Создание управляющих привязок в файле attr.directive.ts

```
import { Directive, ElementRef, Attribute, Input,
         SimpleChange, Output, EventEmitter, HostListener, HostBinding }
            from "@angular/core";
 import { Product } from "./product.model";

@Directive({
    selector: "[pa-attr]"
})
export class PaAttrDirective {

    @Input("pa-attr")
    @HostBinding("class")
    bgClass: string;

    @Input("pa-product")
    product: Product;

    @Output("pa-category")
    click = new EventEmitter<string>();

    @HostListener("click")
    triggerCustomEvent() {
        if (this.product != null) {
            this.click.emit(this.product.category);
        }
    }
}
```

Декоратор `@HostBinding` используется для создания привязки свойства для управляющего элемента и применяется к свойству директивы. В листинге создается привязка между свойством класса управляющего элемента и декоратором свойства `bgClass`.

ПРИМЕЧАНИЕ

Если вы хотите управлять содержимым элемента, используйте декоратор @HostBinding для привязки к свойству textContent. Пример приведен в главе 19.

Декоратор `@HostListener` используется для создания привязки события для управляющего элемента и применяется к методу. В листинге создается привязка для события `click`, которое вызывает метод `triggerCustomEvent` при нажатии и отпускании кнопки мыши. Как подсказывает название, метод `triggerCustomEvent` использует метод `EventEmitter.emit` для передачи нестандартного события через выходное свойство.

Использование привязок управляющих элементов означает, что конструктор директивы можно удалить, так как обращаться к элементу HTML через объект `ElementRef` уже не нужно. Вместо этого Angular создает слушателя события и задает принадлежность элемента к классам через привязку свойства.

Код директивы заметно упростился, но эффект остался неизменным: щелчок на строке таблицы задает значение одного из элементов `input`, а добавление нового объекта с формы инициирует изменение цвета фона ячеек таблицы для товаров, не входящих в категорию `Soccer`.

Создание двусторонней привязки для управляющего элемента

Angular предоставляет специальную поддержку для создания директив, поддерживающих двусторонние привязки, чтобы они могли применяться со скобками `[()]`, используемыми `ngModel`, а привязка к свойствам модели работала в обоих направлениях.

Функциональность двусторонней привязки опирается на соглашения об именах. Чтобы продемонстрировать, как она работает, в листинге 15.16 в файл template.html добавляются новые элементы и привязки.

Листинг 15.16. Применение директивы в файле template.html

```
...
<div class="col-xs-6">

    <div class="form-group bg-info p-a-1">
        <label>Name:</label>
        <input class="bg-primary" [paModel]="newProduct.name"
                                  (paModelChange)="newProduct.name=$event" />
```

```
    </div>

    <table class="table table-sm table-bordered table-striped">
        <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
        <tr *ngFor="let item of getProducts(); let i = index"
                [pa-attr]="getProducts().length < 6 ? 'bg-success' : 'bg-warning'"
                [pa-product]="item" (pa-category)="newProduct.category=$event">
            <td>{{i + 1}}</td>
            <td>{{item.name}}</td>
            <td [pa-attr]="item.category == 'Soccer' ? 'bg-info' : null">
                {{item.category}}
            </td>
            <td [pa-attr]="'bg-info'">{{item.price}}</td>
        </tr>
    </table>
</div>
...
```

Сейчас мы создадим директиву, поддерживающую две односторонние привязки. Привязка, целью которой является `paModel`, обновляется при изменении свойства `newProduct.name`; она обеспечивает передачу данных от приложения к директиве и используется для обновления содержимого элемента `input`. Нестандартное событие `paModelChange` срабатывает, когда пользователь изменяет содержимое элемента `input`, и обеспечивает передачу данных от директивы к остальному коду приложения.

Чтобы реализовать эту директиву, создайте файл twoway.directive.ts в папке app и включите в него определение директивы из листинга 15.17.

Листинг 15.17. Содержимое файла twoway.directive.ts из папки app

```
import { Input, Output, EventEmitter, Directive,
         HostBinding, HostListener, SimpleChange } from "@angular/core";

@Directive({
    selector: "input[paModel]"
})
export class PaModel {

    @Input("paModel")
    modelProperty: string;

    @HostBinding("value")
    fieldValue: string = "";

    ngOnChanges(changes: { [property: string]: SimpleChange }) {
        let change = changes["modelProperty"];
        if (change.currentValue != this.fieldValue) {
            this.fieldValue = changes["modelProperty"].currentValue || "";
        }
    }

    @Output("paModelChange")
```

```
    update = new EventEmitter<string>();

    @HostListener("input", ["$event.target.value"])
    updateValue(newValue: string) {
        this.fieldValue = newValue;
        this.update.emit(newValue);
    }
}
```

Директива использует функциональность, описанную выше. Свойство `selector` этой директивы указывает, что совпадение будет найдено для элементов `input` с атрибутом `paModel`. Встроенная двусторонняя директива `ngModel` поддерживает различные элементы форм и знает, какие события и свойства поддерживаются каждым из них. Однако я хочу сделать свой пример простым, поэтому поддерживаться будут только элементы `input` со свойством `value`, используемым для чтения и записи содержимого элемента.

Привязка `paModel` реализуется на базе входного свойства и метода `ngOnChanges`, который реагирует на изменение значения выражения, обновляя содержимое элемента `input` через привязку к свойству `value` элемента `input`.

Событие `paModelChange` реализуется через слушателя события `input`, который затем отправляет обновление через выходное свойство. Обратите внимание: метод, вызываемый событием, может получить объект события, передавая дополнительный аргумент декоратору `@HostListener`:

```
...
@HostListener("input", ["$event.target.value"])
updateValue(newValue: string) {
...
```

Первый аргумент декоратора `@HostListener` задает имя события, которое будет обрабатываться слушателем. Второй аргумент содержит массив, в котором декорированным методам будут передаваться аргументы. В нашем примере слушателем обрабатывается событие `input`, а при вызове метода `updateValue` аргументу `newValue` задается свойство `target.value` объекта `Event`, для ссылки на которое используется обозначение `$event`.

Чтобы включить поддержку директивы, добавьте ее в модуль Angular (листинг 15.18).

Листинг 15.18. Регистрация директивы в файле app.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { ProductComponent } from "./component";
import { FormsModule, ReactiveFormsModule  } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
import { PaModel } from "./twoway.directive";
@NgModule({
    imports: [BrowserModule, FormsModule, ReactiveFormsModule],
```

```
    declarations: [ProductComponent, PaAttrDirective, PaModel],
    bootstrap: [ProductComponent]
})
export class AppModule { }
```

После сохранения изменений и обновления браузера появляется новый элемент `input`, который реагирует на изменения свойства модели и обновляет его при изменении содержимого управляющего элемента. В выражениях привязок задается то же свойство модели, которое используется в поле Name формы в левой части документа HTML; это упрощает тестирование связи между ними (рис. 15.7).

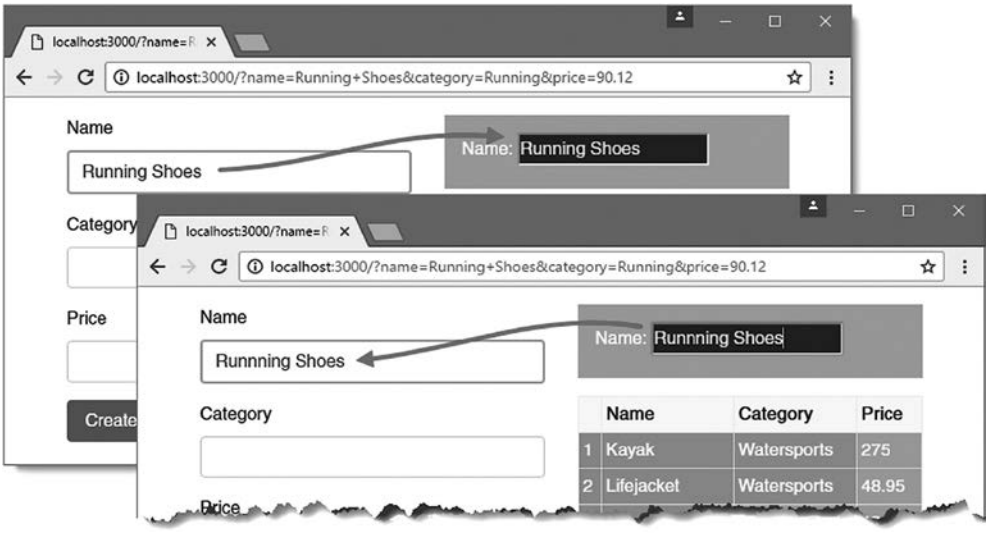


Рис. 15.7. Тестирование двусторонней передачи данных

Остается сделать последний шаг — упростить привязки и применить скобки `[()]`, как показано в листинге 15.19.

Листинг 15.19. Упрощение привязок в файле template.html

```
...
<div class="col-xs-6">

    <div class="form-group bg-info p-a-1">
        <label>Name:</label>
        <input class="bg-primary" [(paModel)]="newProduct.name" />
    </div>

    <table class="table table-sm table-bordered table-striped">
        <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
        <tr *ngFor="let item of getProducts(); let i = index"
```

```
                [pa-attr]="getProducts().length < 6 ? 'bg-success' : 'bg-warning'"
                [pa-product]="item" (pa-category)="newProduct.category=$event">
            <td>{{i + 1}}</td>
            <td>{{item.name}}</td>
            <td [pa-attr]="item.category == 'Soccer' ? 'bg-info' : null">
                {{item.category}}
            </td>
            <td [pa-attr]="'bg-info'">{{item.price}}</td>
        </tr>
    </table>
</div>
...
```

Когда Angular обнаруживает скобки `[()]`, привязка расширяется в соответствии с форматом, использованным в листинге 15.16, с входным свойством `paModel` и настройкой события `paModelChange`. Если директива предоставляет Angular доступ к ним, вы можете использовать скобки `[()]` для упрощения синтаксиса шаблона.

Экспортирование директивы для использования в переменной шаблона

В предыдущих главах мы использовали переменные шаблонов для обращения к функциональности, предоставляемой встроенными директивами (такими, как `ngForm`). Пример элемента из главы 14:

```
...
<form novalidate #form="ngForm" (ngSubmit)="submitForm(form)">
...
```

Переменной шаблона `form` присваивается директива `ngForm`, которая затем используется для работы с результатами проверки данных для формы HTML. Этот пример показывает, как директива может предоставлять доступ к своим свойствам и методам, чтобы они могли использоваться в привязках данных и выражениях.

В листинге 15.20 директива из предыдущего раздела изменяется так, чтобы она предоставляла информацию о том, расширяла ли она текст в своем управляющем элементе.

Листинг 15.20. Экспортирование директивы в файле twoway.directive.ts

```
import { Input, Output, EventEmitter, Directive,
    HostBinding, HostListener, SimpleChange } from "@angular/core";

@Directive({
    selector: "input[paModel]",
```

```
    exportAs: "paModel"
})
export class PaModel {

    direction: string = "None";

    @Input("paModel")
    modelProperty: string;
    @HostBinding("value")
    fieldValue: string = "";

    ngOnChanges(changes: { [property: string]: SimpleChange }) {
        let change = changes["modelProperty"];
        if (change.currentValue != this.fieldValue) {
            this.fieldValue = changes["modelProperty"].currentValue || "";
            this.direction = "Model";
        }
    }

    @Output("paModelChange")
    update = new EventEmitter<string>();

    @HostListener("input", ["$event.target.value"])
    updateValue(newValue: string) {
        this.fieldValue = newValue;
        this.update.emit(newValue);
        this.direction = "Element";
    }
}
```

Свойство `exportAs` декоратора `@Directive` определяет имя, которое будет использоваться для ссылок на директиву в переменных шаблонов. В данном примере в качестве значения `exportAs` используется `paModel`; старайтесь использовать имена, которые ясно показывают, какая директива предоставляет функциональность.

В листинге в директиву добавляется свойство с именем `direction`, которое указывает, в какую сторону передаются данные: от модели к элементу или от элемента к модели.

Используя декоратор `exportAs`, вы предоставляете доступ ко всем методам и свойствам, определяемым директивой, для использования в выражениях шаблонов и привязках данных. Некоторые разработчики снабжают имена методов и свойств, которые не должны использоваться за пределами директивы, префиксом `_` (символ подчеркивания) или применяют ключевое слово `private`. Оно сообщает другим разработчикам, что некоторые методы и свойства не должны использоваться, но Angular не следит за выполнением этой рекомендации.

В листинге 15.21 для экспортируемой функциональности директивы создается переменная шаблона, которая используется в привязке стиля.

Листинг 15.21. Использование экспортируемой функциональности директивы в файле template.html

```
...
<div class="col-xs-6">

    <div class="form-group bg-info p-a-1">
        <label>Name:</label>
        <input class="bg-primary" [(paModel)]="newProduct.name" #paModel="paModel" />
        <div class="bg-primary">Direction: {{paModel.direction}}</div>
    </div>

    <table class="table table-sm table-bordered table-striped">
        <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
        <tr *ngFor="let item of getProducts(); let i = index"
                [pa-attr]="getProducts().length < 6 ? 'bg-success' : 'bg-warning'"
                [pa-product]="item" (pa-category)="newProduct.category=$event">
            <td>{{i + 1}}</td>
            <td>{{item.name}}</td>
            <td [pa-attr]="item.category == 'Soccer' ? 'bg-info' : null">
                {{item.category}}
            </td>
            <td [pa-attr]="'bg-info'">{{item.price}}</td>
        </tr>
    </table>
</div>
...
```

Переменная шаблона называется `paModel`, а ее значением является имя, использованное в свойстве `exportAs` директивы.

```
...
#paModel="paModel"
...
```

ПРИМЕЧАНИЕ

Присваивать одинаковые имена переменной и директиве не обязательно, но это помогает более четко обозначить источник функциональности.

После того как переменная шаблона будет определена, она может использоваться в привязках с интерполяцией или как часть выражения привязки. Я выбрал вариант со строковой интерполяцией, в выражении которого используется значение свойства `direction` директивы.

```
...
<div class="bg-primary">Direction: {{paModel.direction}}</div>
...
```

В результате вы можете наблюдать эффект ввода текста в двух элементах `input`, привязанных к свойству модели `newProduct.name`. При вводе текста в поле, использующее директиву `ngModel`, в поле привязки со строковой интерполяцией

выводится сообщение `Model`. При вводе текста в поле, использующее директиву `paModel`, выводится сообщение `Element` (рис. 15.8).

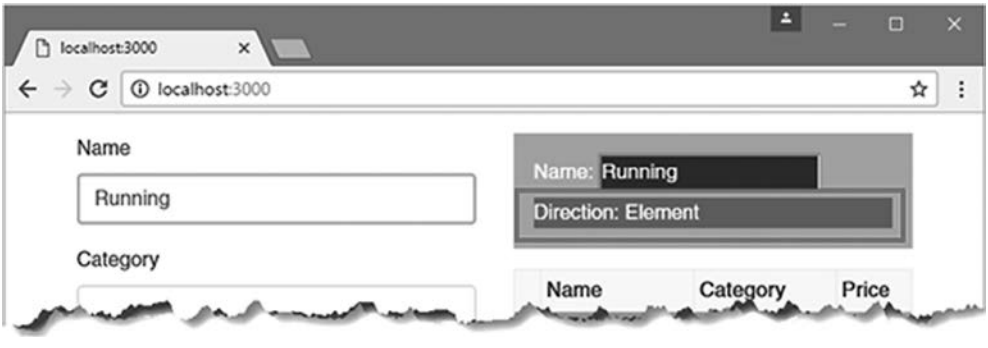


Рис. 15.8. Экспортирование функциональности из директивы

Итоги

В этой главе вы научились определять и использовать директивы атрибутов, в том числе освоили использование входных и выходных свойств и привязок управляющих элементов. В следующей главе я объясню, как работают структурные директивы и как они используются для изменения макета или структуры документов HTML.

16 Создание структурных директив

Структурные директивы изменяют строение документа HTML с добавлением и удалением элементов. Они расширяют базовую функциональность, доступную директивам атрибутов, описанную в главе 15, с дополнительной поддержкой микрошаблонов — небольших фрагментов контента, которые определяются в шаблонах, используемых компонентами. Структурные директивы можно узнать по имени с префиксом * (например, `*ngIf` и `*ngFor`). В этой главе я объясню, как определяются и применяются структурные директивы, как они работают и как реагируют на изменения в модели данных. В табл. 16.1 директивы атрибутов представлены в контексте.

Таблица 16.1. Директивы атрибутов в контексте

Вопрос	Ответ
Что это такое?	Структурные директивы используют микрошаблоны для добавления контента в документ HTML
Для чего они нужны?	Структурные директивы позволяют добавлять контент в зависимости от результата выражения или повторять его для каждого объекта в источнике данных (например, в массиве)
Как они используются?	Структурные директивы применяются к элементу шаблона, который содержит контент и привязки из микрошаблона. Класс шаблона использует объекты, предоставленные Angular, для управления включением или повторением контента
Есть ли у них недостатки или скрытые проблемы?	Если не принять специальных мер, структурные директивы могут вносить в документ HTML множество необязательных изменений, которые отрицательно сказываются на быстродействии веб-приложения. Важно ограничиться только теми изменениями, которые действительно необходимы (см. раздел «Изменения данных на уровне коллекции» этой главы)
Есть ли альтернативы?	Для стандартных задач можно использовать встроенные директивы, но нестандартные структурные директивы позволяют адаптировать поведение под особенности вашего приложения

В табл. 16.2 приведена краткая сводка материала главы.

Таблица 16.2. Сводка материала главы

Проблема	Решение	Листинг
Создание структурной директивы	Примените декоратор @Directive к классу, получающему в параметрах контейнер представлений и конструктор шаблона	1–6
Создание итеративной структурной директивы	Определите входное свойство ForOf в классе структурной директивы и проведите перебор по его значению	7–12
Обработка изменений в данных в структурных директивах	Используйте диффер для обнаружения изменений в методе ngDoCheck	13–19
Получение контента управляющего элемента, к которому применяется структурная директива	Используйте декораторы @ContentChild и @ContentChildren	20–25

Подготовка проекта

В этой главе мы продолжим использовать проект example, созданный в главе 11. Чтобы подготовиться к этой главе, упростите шаблон — удалите из него форму, оставив только таблицу, показанную в листинге 16.1. (Форма будет добавлена позже в этой главе.)

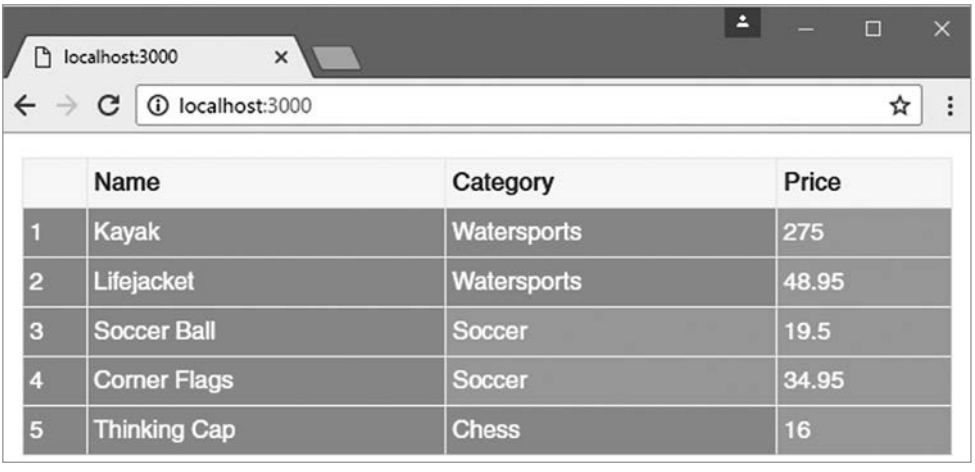
Листинг 16.1. Упрощение шаблона в файле template.html

```
<table class="table table-sm table-bordered table-striped">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
    <tr *ngFor="let item of getProducts(); let i = index"
            [pa-attr]="getProducts().length < 6 ? 'bg-success' : 'bg-warning'"
            [pa-product]="item" (pa-category)="newProduct.category=$event">
        <td>{{i + 1}}</td>
        <td>{{item.name}}</td>
        <td [pa-attr]="item.category == 'Soccer' ? 'bg-info' : null">
            {{item.category}}
        </td>
        <td [pa-attr]="'bg-info'">{{item.price}}</td>
    </tr>
</table>
```

Выполните следующую команду из папки example, чтобы запустить сервер для разработки и компилятор TypeScript и загрузить приложение Angular в браузере:

```
npm start
```

На рис. 16.1 показан исходный контент, отображаемый в браузере.



	Name	Category	Price
1	Kayak	Watersports	275
2	Lifejacket	Watersports	48.95
3	Soccer Ball	Soccer	19.5
4	Corner Flags	Soccer	34.95
5	Thinking Cap	Chess	16

Рис. 16.1. Запуск приложения

Создание простой структурной директивы

Знакомство со структурными директивами лучше всего начать с воссоздания функциональности директивы `ngIf` — она относительно проста и помогает понять, как работают структурные директивы.

Для начала внесем изменения в шаблон и вернемся к коду, который обеспечивает его работу. В листинге 16.2 показаны изменения в шаблоне.

Листинг 16.2. Применение структурной директивы в файле template.html

```
<div class="checkbox">
    <label>
        <input type="checkbox" [(ngModel)]="showTable" />
        Show Table
    </label>
</div>

<template [paIf]="showTable">
    <table class="table table-sm table-bordered table-striped">
        <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
        <tr *ngFor="let item of getProducts(); let i = index"
            [pa-attr]="getProducts().length < 6 ? 'bg-success' : 'bg-warning'"
            [pa-product]="item" (pa-category)="newProduct.category=$event">
            <td>{{i + 1}}</td>
            <td>{{item.name}}</td>
            <td [pa-attr]="item.category == 'Soccer' ? 'bg-info' : null">
                {{item.category}}
            </td>
            <td [pa-attr]="'bg-info'">{{item.price}}</td>
        </tr>
    </table>
</template>
```

В листинге используется полный синтаксис шаблона, в котором директива применяется к элементу шаблона, содержащему контент, используемый директивой. В нашем примере элемент шаблона содержит элемент таблицы и весь его контент, включая привязки, директивы и выражения. Также существует компактный синтаксис, который будет использоваться позднее в этой главе.

Элемент шаблона имеет стандартную одностороннюю привязку данных, целью которой является директива с именем `paIf`:

```
...
<template [paIf]="showTable">
...
```

В выражении этой привязки используется значение свойства с именем `showTable`. Это же свойство используется в другой привязке шаблона, которая применяется к элементу флажка:

```
...
<input type="checkbox" checked="true" [(ngModel)]="showTable" />
...
```

В этом разделе мы хотим создать структурную директиву, которая добавит содержимое элемента шаблона в документ HTML, если свойство `showTable` истинно; это условие выполняется, если флажок установлен. Если же свойство `showTable` ложно (это происходит, если флажок снят), то содержимое элемента шаблона исключается.

Реализация класса структурной директивы

Из шаблона понятно, что должна делать директива. Чтобы реализовать директиву, создайте файл paif.directive.ts в папке app и добавьте код из листинга 16.3.

Листинг 16.3. Содержимое файла structure.directive.ts в папке app

```
import {
    Directive, SimpleChange, ViewContainerRef, TemplateRef, Input
} from "@angular/core";

@Directive({
    selector: "[paIf]"
})
export class PaStructureDirective {

    constructor(private container: ViewContainerRef,
        private template: TemplateRef<Object>) { }

    @Input("paIf")
    expressionResult: boolean;

    ngOnChanges(changes: { [property: string]: SimpleChange }) {
        let change = changes["expressionResult"];
```

```
        if (!change.isFirstChange() && !change.currentValue) {
            this.container.clear();
        } else if (change.currentValue) {
            this.container.createEmbeddedView(this.template);
        }
    }
}
```

Свойство `selector` декоратора `@Directive` используется для отбора управляющих элементов с атрибутом `paIf` (в соответствии с изменениями шаблона, внесенными в листинге 16.1).

Входное свойство с именем `expressionResult` используется директивой для получения результатов выражения из шаблона. Директива реализует метод `ngOnChanges` для получения уведомлений об изменениях, чтобы она могла реагировать на изменения в модели данных.

Первый признак того, что директива является структурной, встречается в конструкторе, который требует у Angular предоставить параметры с новыми типами.

```
...
constructor(private container: ViewContainerRef,
      private template: TemplateRef<Object>) {}
...
```

Объект `ViewContainerRef` используется для управления содержимым *контейнера представлений* — части документа HTML, в которой отображается элемент `template` и за которую отвечает директива.

Как подсказывает название, контейнер представлений отвечает за управление коллекцией *представлений*. Представление (view) — это набор элементов HTML, содержащий директивы, привязки и выражения; для создания и управления ими используются методы и свойства, предоставленные классом `ViewContainerRef`. Самые полезные из этих методов и свойств перечислены в табл. 16.3.

Таблица 16.3. Важнейшие методы и свойства ViewContainerRef

Имя	Описание
element	Свойство возвращает элемент ElementRef, представляющий контейнерный элемент
createEmbeddedView (template)	Метод использует шаблон для создания нового представления (подробности в тексте после таблицы). Метод также получает необязательные аргументы с данными контекста (см. раздел «Создание итеративных структурных директив») и индексом, который указывает, где должно быть вставлено представление. В результате создается объект ViewRef, который может использоваться с другими методами этой таблицы
clear()	Метод удаляет все представления из контейнера
length	Свойство возвращает количество представлений в контейнере
get(index)	Метод возвращает объект ViewRef для представления с заданным индексом

Имя	Описание
indexOf(view)	Метод возвращает индекс заданного объекта ViewRef
insert(view, index)	Метод вставляет представление с заданным индексом
remove(Index)	Метод удаляет и уничтожает представление с заданным индексом
detach(index)	Метод отсоединяет представление с заданным индексом без его уничтожения для последующего изменения его позиции методом insert

Для воссоздания функциональности директивы `ngIf` необходимы два метода из перечисленных в табл. 16.3: метод `createEmbeddedView` для вывода содержимого элемента `template` и метод `clear` для его повторного удаления.

Метод `createEmbeddedView` добавляет представление в контейнер представлений. Аргумент метода содержит объект `TemplateRef`, который представляет содержимое элемента `template`.

Директива получает объект `TemplateRef` в одном из аргументов конструктора; Angular автоматически предоставляет его значение при создании нового экземпляра класса директивы.

Если объединить все сказанное воедино, становится ясно, как работает директива. Обрабатывая файл template.html, Angular обнаруживает элемент `template` с привязкой и определяет, что нужно создать новый экземпляр класса `PaStructureDirective`. Angular анализирует конструктор `PaStructureDirective` и видит, что ему нужно предоставить объекты `ViewContainerRef` и `TemplateRef`.

```
...
constructor(private container: ViewContainerRef,
    private template: TemplateRef<Object>) {}
...
```

Объект `ViewContainerRef` представляет место в документе HTML, занимаемое элементом `template`, а `TemplateRef` представляет содержимое элемента `template`. Angular передает эти объекты конструктору и создает новый экземпляр класса директивы.

Затем Angular начинает обрабатывать выражения и привязки данных. Как было описано в главе 15, Angular вызывает метод `ngOnChanges` в ходе инициализации (непосредственно перед вызовом метода `ngOnInit`), а потом повторно при изменении значения выражения директивы.

Реализация метода `ngOnChanges` в классе `PaStructureDirective` использует получаемый объект `SimpleChange` для отображения или сокрытия содержимого элемента `template` в зависимости от текущего значения выражения. Если выражение истинно, директива выводит содержимое элемента `template`, добавляя его в контейнер представлений.

```
...
this.container.createEmbeddedView(this.template);
...
```

Если результат выражения равен `false`, директива очищает контейнер представлений, что приводит к удалению элементов из документа HTML.

```
...
this.container.clear();
...
```

Директива ничего не знает о содержимом элемента `template`; она отвечает только за управление его видимостью.

Регистрация структурной директивы

Чтобы директиву можно было использовать в приложении, ее следует включить в модуль Angular (листинг 16.4).

Листинг 16.4. Регистрация директивы в файле app.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { ProductComponent } from "./component";
import { FormsModule, ReactiveFormsModule  } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
import { PaModel } from "./twoway.directive";
import { PaStructureDirective } from "./structure.directive";

@NgModule({
    imports: [BrowserModule, FormsModule, ReactiveFormsModule],
    declarations: [ProductComponent, PaAttrDirective, PaModel,
PaStructureDirective],
    bootstrap: [ProductComponent]
})
export class AppModule { }
```

Структурные директивы включаются так же, как и атрибуты директив, и задаются в массиве `declarations` модуля.

После сохранения изменений браузер перезагружает документ HTML, и вы немедленно видите эффект новой директивы: элемент `table`, который является содержимым элемента `template`, отображается только при установленном флажке (рис. 16.2).

ПРИМЕЧАНИЕ

Содержимое элемента template уничтожается и создается заново, а не просто скрывается и отображается заново. Если вы хотите скрыть или отобразить контент без удаления его из документа HTML, используйте привязку стиля для управления свойством display или visibility.

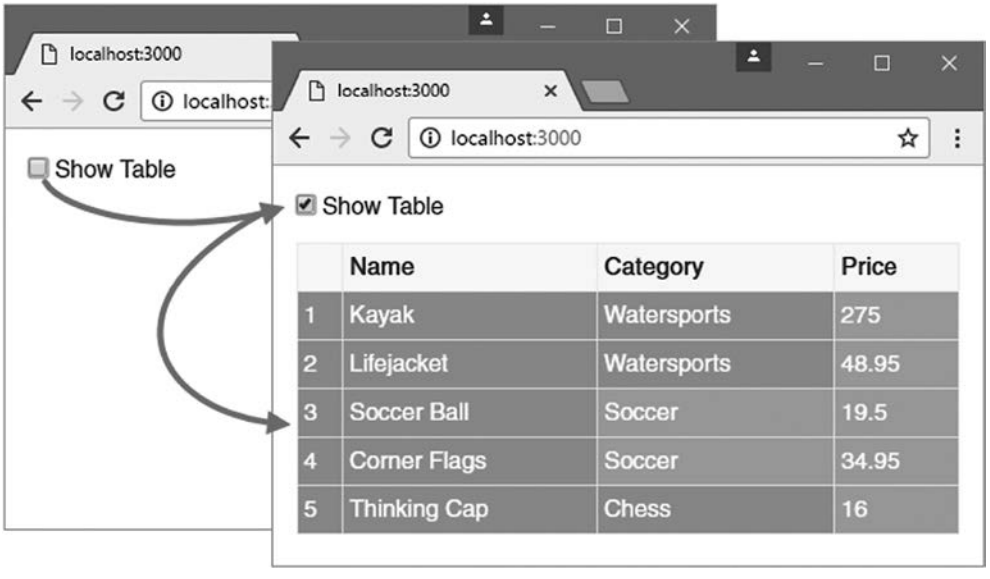


Рис. 16.2. Создание структурной директивы

Определение исходного значения выражения

Содержимое элемента `template` изначально не выводится, потому что выражение директивы зависит от переменной, которая не была определена ранее.

```
...
<template [paIf]="showTable">
...
```

Переменная `showTable` не будет определена до того, как пользователь установит флажок; в этот момент она будет создана со значением `true`. Это пример того, как шаблоны могут динамически создавать переменные JavaScript (см. главу 14). Я упоминаю об этом здесь, потому что легко запутаться в том, где создается переменная. Если вы хотите определить исходное значение переменной `showTable`, это необходимо сделать в компоненте, как показано в листинге 16.5, а не в структурной директиве.

Листинг 16.5. Определение переменной в файле component.ts

```
import { ApplicationRef, Component } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";
import { ProductFormGroup } from "./form.model";

@Component({
    selector: "app",
    templateUrl: "app/template.html"
```

```
})
export class ProductComponent {
    model: Model = new Model();
    form: ProductFormGroup = new ProductFormGroup();

    // ...Другие члены класса опущены для краткости...

    showTable: boolean = true;
}
```

Структурная директива получает значение своего выражения через входное свойство и ничего не знает об этом выражении; компонент предоставляет Angular контекст для вычисления выражения. Изменение из листинга 16.5 задает свойству `showTable` начальное значение `true`; это означает, что содержимое элемента `template` будет отображаться при запуске приложения.

Компактный синтаксис структурных директив

Использование элемента `template` помогает продемонстрировать роль контейнера представлений в структурных директивах. Компактный синтаксис избавляется от элемента `template`; директива и ее выражение применяются к внешнему содержащему элементу, как показано в листинге 16.6.

ПРИМЕЧАНИЕ

Компактный синтаксис структурных директив был создан для того, чтобы упростить их чтение и использование. Тем не менее выбор синтаксиса зависит исключительно от личных предпочтений.

Листинг 16.6. Использование компактного синтаксиса структурных директив в файле template.html

```
<div class="checkbox">
    <label>
        <input type="checkbox" [(ngModel)]="showTable" />
        Show Table
    </label>
</div>

<table *paIf="showTable"
        class="table table-sm table-bordered table-striped">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
    <tr *ngFor="let item of getProducts(); let i = index"
        [pa-attr]="getProducts().length < 6 ? 'bg-success' : 'bg-warning'"
        [pa-product]="item" (pa-category)="newProduct.category=$event">
        <td>{{i + 1}}</td>
        <td>{{item.name}}</td>
        <td [pa-attr]="item.category == 'Soccer' ? 'bg-info' : null">
            {{item.category}}
        </td>
        <td [pa-attr]="'bg-info'">{{item.price}}</td>
    </tr>
</table>
```

Элемент `template` удален, а директива применяется к элементу `table`:

```
...
<table *paIf="showTable" class="table table-sm table-bordered table-striped">
...
```

Имя директивы снабжается префиксом * (звездочка), который сообщает Angular, что это структурная директива, использующая компактный синтаксис. Разбирая файл template.html, Angular обнаруживает директиву и звездочку и обрабатывает элементы так, словно в документе присутствует элемент `template`. Поддержка компактного синтаксиса не требует никаких изменений в классе директивы.

Создание итеративных структурных директив

Angular предоставляет особые средства для директив, которые должны перебирать содержимое источника данных. Чтобы продемонстрировать их использование, лучше всего воссоздать функциональность другой из встроенных директив: `ngFor`.

Чтобы подготовиться к использованию новой директивы, удалите директиву `ngFor` из файла template.html, вставьте элемент `template` и примените новую директиву атрибута и выражение, как показано в листинге 16.7.

Листинг 16.7. Подготовка к использованию новой структурной директивы в файле template.html

```
<div class="checkbox">
    <label>
        <input type="checkbox" [(ngModel)]="showTable" />
        Show Table
    </label>
</div>
<table *paIf="showTable"
        class="table table-sm table-bordered table-striped">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
    <template [paForOf]="getProducts()" let-item>
        <tr><td colspan="4">{{item.name}}</td></tr>
    </template>
</table>
```

Полный синтаксис итеративных структурных директив выглядит немного странно. В листинге элемент `template` содержит два атрибута, использованные для применения директивы. Первый — стандартная привязка, выражение которой получает необходимые директиве данные для атрибута `paForOf`.

```
...
<template [paForOf]="getProducts()" let-item>
...
```

Имя атрибута важно. При использовании элемента `template` имя атрибута источника данных должно заканчиваться символами `Of` для поддержки компактного синтаксиса, который будет описан ниже.

Второй атрибут используется для определения *неявного значения* (implicit value), которое позволяет ссылаться на текущий обрабатываемый объект из элемента `template` во время перебора источника данных директивой. В отличие от других переменных шаблонов, неявной переменной не присваивается значение, а существует она только ради определения имени переменной.

```
...
<template [paForOf]="getProducts()" let-item>
...
```

В этом примере конструкция `let-item` сообщает Angular, что неявное значение должно быть присвоено переменной с именем `item`, которая затем используется в привязке со строковой интерполяцией для отображения свойства `name` текущего объекта данных.

```
...
<td colspan="4">{{item.name}}</td>
...
```

Как видно из элемента `template`, новая директива предназначена для перебора результатов метода `getProducts` компонента, и для каждого объекта она генерирует строку таблицы со свойством `name`. Для реализации этой функциональности создайте файл iterator.directive.ts в папке app и определите директиву из листинга 16.8.

Листинг 16.8. Содержимое файла iterator.directive.ts в папке app

```
import { Directive, ViewContainerRef, TemplateRef,
             Input, SimpleChange } from "@angular/core";

@Directive({
    selector: "[paForOf]"
})
export class PaIteratorDirective {

    constructor(private container: ViewContainerRef,
        private template: TemplateRef<Object>) {}
    @Input("paForOf")
    dataSource: any;

    ngOnInit() {
        this.container.clear();
        for (let i = 0; i < this.dataSource.length; i++) {
            this.container.createEmbeddedView(this.template,
                 new PaIteratorContext(this.dataSource[i]));
        }
    }
}

class PaIteratorContext {
    constructor(public $implicit: any) {}
}
```

Свойство `selector` в декораторе `@Directive` выбирает элементы с атрибутом `paForOf`, которые также являются источником данных для входного свойства `dataSource` и которые предоставляют источник перебираемых объектов.

После задания значения входного свойства будет вызван метод `ngOnInit`, директива очищает контейнер представлений методом `clear` и добавляет новое представление для каждого объекта методом `createEmbeddedView`.

При вызове метода `createEmbeddedView` директива предоставляет два аргумента: объект `TemplateRef`, полученный через конструктор, и объект контекста. Объект `TemplateRef` предоставляет контент для вставки в контейнер, а объект контекста предоставляет данные для неявного значения, которое задается при помощи свойства `$implicit`. Этот объект со свойством `$implicit` присваивается переменной шаблона `item`, и на него ссылается привязка со строковой интерполяцией. Чтобы предоставить шаблонам объект контекста способом, безопасным по отношению к типам, я определил класс `PaIteratorContext`, единственное свойство которого называется `$implicit`.

Метод `ngOnInit` раскрывает некоторые важные аспекты работы с контейнерами представлений. Во-первых, контейнер представлений может быть заполнен несколькими представлениями — в данном случае по одному представлению на объект в источнике данных. Класс `ViewContainerRef` предоставляет функциональность, необходимую для управления представлениями после их создания (см. далее).

Во-вторых, шаблон может повторно использоваться для создания нескольких представлений. В этом примере содержимое элемента `template` будет использоваться для создания идентичных элементов `tr` и `td` для каждого объекта в источнике данных. Элемент `td` содержит привязку данных, которая обрабатывается Angular при создании каждого представления и используется для адаптации контента к объекту данных.

В-третьих, директива не обладает информацией о данных, с которыми она работает, и о генерируемом контенте. Angular предоставляет директиве необходимый контекст в других частях приложения; источник данных предоставляется через входное свойство, а контент для каждого представления — через объект `TemplateRef`.

Чтобы активизировать директиву, следует добавить новую запись в модуль Angular (листинг 16.9).

Листинг 16.9. Добавление нестандартной директивы в файл app.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { ProductComponent } from "./component";
import { FormsModule, ReactiveFormsModule  } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
import { PaModel } from "./twoway.directive";
import { PaStructureDirective } from "./structure.directive";
import { PaIteratorDirective } from "./iterator.directive";
```

```
@NgModule({
    imports: [BrowserModule, FormsModule, ReactiveFormsModule],
    declarations: [ProductComponent, PaAttrDirective, PaModel,
        PaStructureDirective, PaIteratorDirective],
    bootstrap: [ProductComponent]
})
export class AppModule { }
```

В результате директива перебирает объекты в источнике данных и использует контент элемента `template` для создания представления для каждого объекта; так строятся строки таблицы (рис. 16.3).

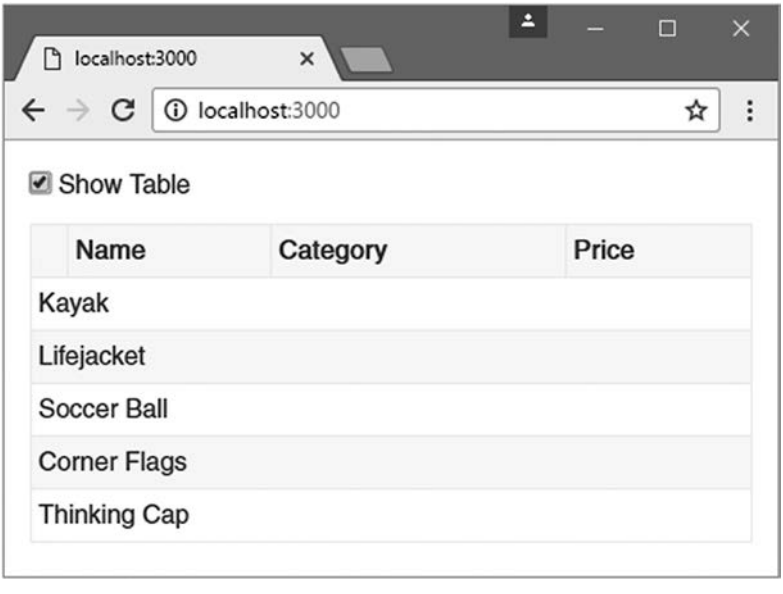


Рис. 16.3. Создание итеративной структурной директивы

Предоставление дополнительных контекстных данных

Структурные директивы могут предоставлять шаблонам дополнительные значения, которые присваиваются переменным шаблонов и используются в привязках. Например, директива `ngFor` предоставляет значения `odd`, `even`, `first` и `last`. Контекстные значения предоставляются через тот же объект, который определяет свойство `$implicit`; в листинге 16.10 воссоздается тот же набор значений, который предоставляет `ngFor`.

Листинг 16.10. Предоставление контекстных данных в файле iterator.directive.ts

```
import { Directive, ViewContainerRef, TemplateRef,
             Input, SimpleChange } from "@angular/core";

@Directive({
```

```
    selector: "[paForOf]"
})

export class PaIteratorDirective {

    constructor(private container: ViewContainerRef,
        private template: TemplateRef<Object>) {}

    @Input("paForOf")
    dataSource: any;

    ngOnInit() {
        this.container.clear();
        for (let i = 0; i < this.dataSource.length; i++) {
            this.container.createEmbeddedView(this.template,
                 new PaIteratorContext(this.dataSource[i],
                     i, this.dataSource.length));
        }
    }
}

class PaIteratorContext {
    odd: boolean; even: boolean;
    first: boolean; last: boolean;

    constructor(public $implicit: any,
            public index: number, total: number ) {

        this.odd = index % 2 == 1;
        this.even = !this.odd;
        this.first = index == 0;
        this.last = index == total - 1;
    }
}
```

В листинге определяются дополнительные свойства в классе `PaIteratorContext`, а конструктор расширяется для получения дополнительных параметров, которые используются для задания значений свойств.

Все эти изменения приводят к тому, что свойства объекта контекста могут использоваться для создания переменных шаблона, ссылки на которые затем включаются в выражения привязки (листинг 16.11).

Листинг 16.11. Использование контекстных данных структурной директивы в файле template.html

```
<div class="checkbox">
    <label>
        <input type="checkbox" [(ngModel)]="showTable" />
        Show Table
    </label>
</div>
```

```
<table *paIf="showTable"
    class="table table-sm table-bordered table-striped">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
    <template [paForOf]="getProducts()" let-item let-i="index"
            let-odd="odd" let-even="even">
        <tr [class.bg-info]="odd" [class.bg-warning]="even">
            <td>{{i + 1}}</td>
            <td>{{item.name}}</td>
            <td>{{item.category}}</td>
            <td>{{item.price}}</td>
        </tr>
    </template>
</table>
```

Переменные шаблонов создаются синтаксисом `let-`<*имя*>, и им присваивается одно из контекстных значений. В этом листинге контекстные значения `odd` и `even` используются для создания одноименных переменных шаблона, которые затем встраиваются в привязки класса в элементе `tr`; в результате строки таблицы окрашиваются с чередованием цветов (рис. 16.4). Листинг также добавляет дополнительные ячейки таблицы для отображения всех свойств `Product`.

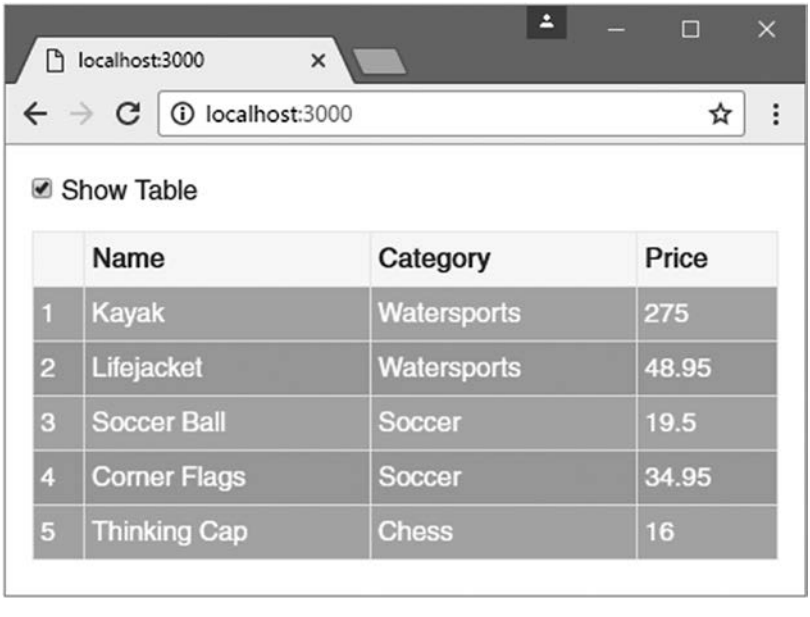


Рис. 16.4. Использование контекстных данных директивы

Компактный структурный синтаксис

Итеративные структурные директивы поддерживают компактный синтаксис без элемента `template` (листинг 16.12).

Листинг 16.12. Использование компактного синтаксиса в файле template.html

```
<div class="checkbox">
    <label>
        <input type="checkbox" [(ngModel)]="showTable" />
        Show Table
    </label>
</div>
<table *paIf="showTable"
    class="table table-sm table-bordered table-striped">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
    <tr *paFor="let item of getProducts(); let i = index; let odd = odd;
            let even = even" [class.bg-info]="odd" [class.bg-warning]="even">
        <td>{{i + 1}}</td>
        <td>{{item.name}}</td>
        <td>{{item.category}}</td>
        <td>{{item.price}}</td>
    </tr>
</table>
```

Здесь требуются более серьезные изменения, чем для директив атрибутов. Самое значительное изменение — атрибут, используемый для применения директивы. При использовании полного синтаксиса директива применялась к элементу `template` с использованием атрибута, заданного селектором:

```
...
<template [paForOf]="getProducts()" let-item let-i="index" let-odd="odd"
    let-even="even">
...
```

При использовании компактного синтаксиса часть `Of` атрибута опускается, имя снабжается префиксом *, а скобки опускаются.

```
...
<tr *paFor="let item of getProducts(); let i = index; let odd = odd;
            let even = even" [class.bg-info]="odd" [class.bg-warning]="even">
...
```

Другое изменение — включение всех контекстных значений в выражение директивы с заменой отдельных атрибутов `let-`. Главное значение данных становится частью исходного выражения, с разделением дополнительных контекстных значений символами `;`.

Для поддержки компактного синтаксиса, в селекторе и входном свойстве которого задается атрибут с именем `paForOf`, никакие изменения в директиве не потребуются. Angular берет на себя расширение компактного синтаксиса, и директива не знает, используется элемент `template` или нет (да для нее это и не важно).

Изменения данных на уровне свойств

В источниках данных, используемых итеративными структурными директивами, возможны изменения двух типов. Первый тип встречается при изменении свойств отдельных объектов. Он порождает каскадный эффект в привязках данных, содержащихся в элементе `template`, — либо напрямую через изменение неявного

значения, либо косвенно через дополнительные контекстные значения, предоставленные директивой. Angular обрабатывает такие изменения автоматически, а все изменения отражаются в контекстных данных привязок, зависящих от них.

Для демонстрации в листинге 16.13 в конструкторе класса контекста добавляется вызов стандартной функции JavaScript `setInterval`. Функция, передаваемая `setInterval`, переключает свойства `odd` и `even`, а также изменяет значение свойства `price` объекта `Product`, который используется как неявное значение.

Листинг 16.13. Изменение отдельных объектов в файле iterator.directive.ts

```
...
class PaIteratorContext {
    odd: boolean; even: boolean;
    first: boolean; last: boolean;

    constructor(public $implicit: any,
            public index: number, total: number ) {

        this.odd = index % 2 == 1;
        this.even = !this.odd;
        this.first = index == 0;
        this.last = index == total - 1;

        setInterval(() => {
            this.odd = !this.odd; this.even = !this.even;
            this.$implicit.price++;
        }, 2000);
    }
}
...
```

Через каждые две секунды значения свойств `odd` и `even` инвертируются, а значение `price` увеличивается. Сохранив изменения, вы увидите, что цвета строк таблицы изменяются, а цены медленно растут (рис. 16.5).

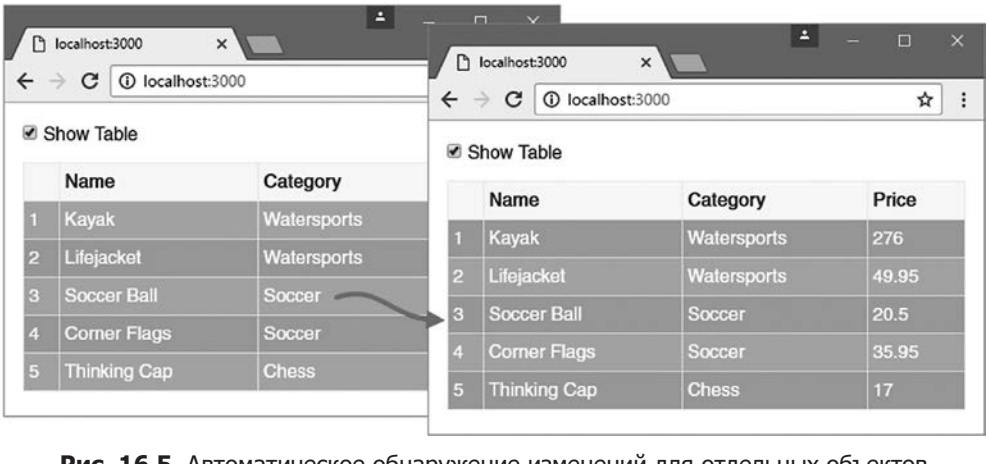


Рис. 16.5. Автоматическое обнаружение изменений для отдельных объектов источника данных

Изменения данных на уровне коллекции

Изменения второго типа происходят при добавлении, удалении или замене объектов коллекции. Angular не обнаруживает такие изменения автоматически, а это означает, что метод `ngOnChanges` итеративной директивы вызываться не будет.

Получение уведомлений об изменениях на уровне коллекции осуществляется реализацией метода `ngDoCheck`, который вызывается при каждом изменении данных в приложении независимо от того, где произошло изменение или к какому типу оно относится. Метод `ngDoCheck` позволяет директиве реагировать на изменения даже в том случае, если они не обнаруживаются Angular автоматически. Однако реализация метода `ngDoCheck` требует внимательности, потому что она может ухудшить быстродействие веб-приложения. Чтобы продемонстрировать суть проблемы, в листинге 16.14 реализуется метод `ngDoCheck`, чтобы директива обновляла отображаемый контент при обнаружении изменений.

Листинг 16.14. Реализация методов ngDoCheck в файле iterator.directive.ts

```
import { Directive, ViewContainerRef, TemplateRef,
             Input, SimpleChange } from "@angular/core";

@Directive({
    selector: "[paForOf]"
})
export class PaIteratorDirective {

    constructor(private container: ViewContainerRef,
        private template: TemplateRef<Object>) {}

    @Input("paForOf")
    dataSource: any;

    ngOnInit() {
        this.updateContent();
    }

    ngDoCheck() {
        console.log("ngDoCheck Called");
        this.updateContent();
    }

    private updateContent() {
        this.container.clear();
        for (let i = 0; i < this.dataSource.length; i++) {
            this.container.createEmbeddedView(this.template,
                 new PaIteratorContext(this.dataSource[i],
                     i, this.dataSource.length));
        }
    }
}

class PaIteratorContext {
    odd: boolean; even: boolean;
```

```
    first: boolean; last: boolean;

    constructor(public $implicit: any,
            public index: number, total: number ) {

       this.odd = index % 2 == 1;
        this.even = !this.odd;
        this.first = index == 0;
        this.last = index == total - 1;

        // setInterval(() => {
        //     this.odd = !this.odd; this.even = !this.even;
        //     this.$implicit.price++;
        // }, 2000);
    }
}
```

Методы `ngOnInit` и `ngDoCheck` вызывают новый метод `updateContent`, который стирает содержимое контейнера представлений и генерирует новый контент шаблона для каждого объекта в источнике данных. При этом вызов функции `setInterval` в классе `PaIteratorContext` закомментирован.

Чтобы понять суть проблемы с изменениями на уровне коллекции и методом `ngDoCheck`, необходимо восстановить форму в шаблоне компонента (листинг 16.15).

Листинг 16.15. Восстановление формы HTML в файле template.html

```
<style>
    input.ng-dirty.ng-invalid { border: 2px solid #ff0000 }
    input.ng-dirty.ng-valid { border: 2px solid #6bc502 }
</style>

<div class="col-xs-4">
    <form novalidate [formGroup]="form" (ngSubmit)="submitForm(form)">
        <div class="form-group" *ngFor="let control of form.productControls">
            <label>{{control.label}}</label>
            <input class="form-control"
                [(ngModel)]="newProduct[control.modelProperty]"
                name="{{control.modelProperty}}"
                formControlName="{{control.modelProperty}}" />
            <ul class="text-danger list-unstyled"
                    *ngIf="(formSubmitted || control.dirty) && !control.valid">
                <li *ngFor="let error of control.getValidationMessages()">
                    {{error}}
                </li>
            </ul>
        </div>
        <button class="btn btn-primary" type="submit"
            [disabled]="formSubmitted && !form.valid"
            [class.btn-secondary]="formSubmitted && !form.valid">
                Create
        </button>
```

```
    </form>
</div>

<div class="col-xs-8">
    <div class="checkbox">
        <label>
            <input type="checkbox" [(ngModel)]="showTable" />
            Show Table
        </label>
    </div>

    <table *paIf="showTable"
        class="table table-sm table-bordered table-striped">
        <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
        <tr *paFor="let item of getProducts(); let i = index; let odd = odd;
                let even = even" [class.bg-info]="odd" [class.bg-warning]="even">
            <td>{{i + 1}}</td>
            <td>{{item.name}}</td>
            <td>{{item.category}}</td>
            <td>{{item.price}}</td>
        </tr>
    </table>
</div>
```

При сохранении изменений в шаблоне форма HTML отображается рядом с таблицей товаров (рис. 16.6).

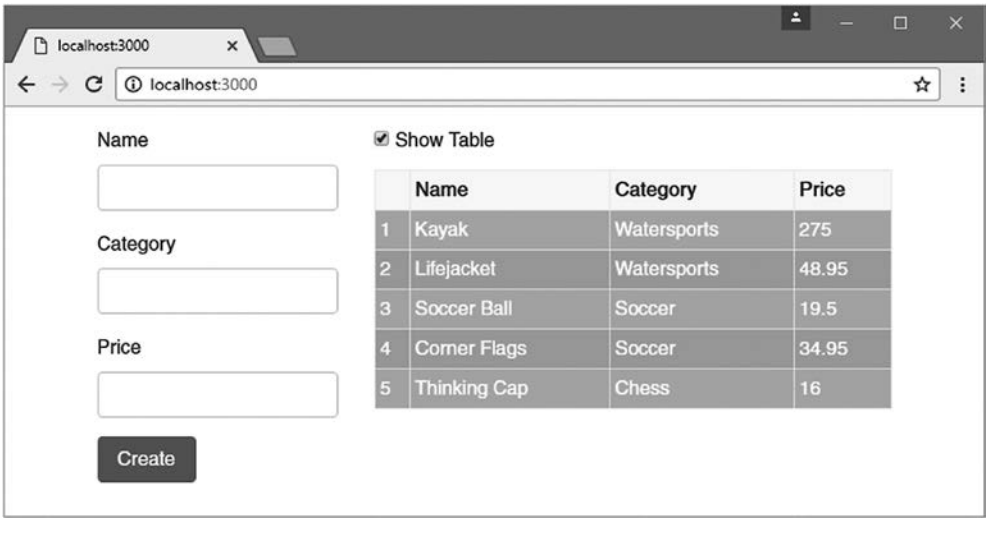


Рис. 16.6. Восстановление таблицы в шаблоне

Проблема с методом `ngDoCheck` заключается в том, что он вызывается каждый раз, когда Angular обнаруживает какие-либо изменения в приложении, — а такие изменения происходят чаще, чем можно было бы предположить.

Чтобы показать, с какой частотой происходят изменения, я добавил вызов метода `console.log` в метод `ngDoCheck` директивы из листинга 16.14, чтобы при каждом вызове метода `ngDoCheck` на консоль JavaScript выводилось сообщение.

Теперь создайте новый товар на форме HTML и посмотрите, сколько сообщений выводится на консоль JavaScript. Каждое сообщение представляет изменение, обнаруженное Angular и приводящее к вызову метода `ngDoCheck`.

Новое сообщение выводится каждый раз, когда элемент `input` получает фокус, при каждом срабатывании события клавиш, при каждом выполнении проверки данных и т. д. Быстрый тест с добавлением в категорию `Running` товара генерирует в моей системе 29 сообщений, хотя точное количество зависит от порядка перемещения между элементами, необходимости исправления опечаток и т. д.

Для каждого из этих 29 раз структурная директива уничтожает и создает заново свой контент; это означает создание новых элементов `tr` и `td`, с новыми директивами и объектами привязки.

Количество записей данных в этом примере невелико, но это высокозатратные операции, и в реальных приложениях многократное уничтожение и повторное создание контента может застопорить работу приложения. А самое неприятное, что все изменения, кроме одного, совершенно излишни, потому что контент таблицы не нужно обновлять до добавления нового объекта `Product` в модель данных. Что касается остальных 28 раз — директива уничтожает свой контент и создает идентичную замену.

К счастью, Angular предоставляет инструменты для более эффективного управления обновлениями, которые позволяют обновлять контент только тогда, когда это необходимо. Метод `ngDoCheck` по-прежнему будет вызываться для всех изменений в приложении, но директива может проанализировать свои данные и определить, произошли ли изменения, требующие нового контента (листинг 16.16).

Листинг 16.16. Минимизация изменения контента в файле iterator.directive.ts

```
import { Directive, ViewContainerRef, TemplateRef,
             Input, SimpleChange, IterableDiffer, IterableDiffers,
             ChangeDetectorRef, CollectionChangeRecord } from "@angular/core";

@Directive({
    selector: "[paForOf]"
})
export class PaIteratorDirective {
    private differ: IterableDiffer;

    constructor(private container: ViewContainerRef,
        private template: TemplateRef<Object>,
        private differs: IterableDiffers,
        private changeDetector: ChangeDetectorRef) {
    }

    @Input("paForOf")
    dataSource: any;
```

```
    ngOnInit() {
        this.differ = this.differs.find(this.dataSource).create
                      (this.changeDetector);
    }

    ngDoCheck() {
        let changes = this.differ.diff(this.dataSource);
        if (changes != null) {
            console.log("ngDoCheck called, changes detected");
            changes.forEachAddedItem(addition => {
                this.container.createEmbeddedView(this.template,
                     new PaIteratorContext(addition.item,
                         addition.currentIndex, changes.length));
            });
        }
    }
}

class PaIteratorContext {
    odd: boolean; even: boolean;
    first: boolean; last: boolean;

    constructor(public $implicit: any,
            public index: number, total: number ) {

        this.odd = index % 2 == 1;
        this.even = !this.odd;
        this.first = index == 0;
        this.last = index == total - 1;
    }
}
```

Идея в том, чтобы определить, были ли объекты добавлены, удалены или перемещены из коллекции. Это означает, что директива должна выполнять некоторую работу каждый раз, когда вызывается метод `ngDoCheck`; это делается для того, чтобы избежать излишних высокозатратных операций DOM при отсутствии изменений в коллекциях, требующих обработки.

Процесс начинается в конструкторе, который получает два новых аргумента, значения которых будут предоставляться Angular при создании нового экземпляра класса директивы. Объекты `IterableDiffers` и `ChangeDetectorRef` используются для организации обнаружения изменений в коллекции источника данных в методе `ngOnInit`:

```
...
ngOnInit() {
    this.differ = this.differs.find(this.dataSource).create(this.changeDetector);
}
...
```

Angular включает встроенные классы, называемые *дифферами* (differs) и предназначенные для обнаружения изменений в разных типах объектов. Метод `Itera-`

bleDiffers.find получает объект и возвращает объект IterableDifferFactory, умеющий создавать класс диффера для этого объекта. Класс IterableDifferFactory определяет метод create, который возвращает объект IterableDiffer для выполнения фактического обнаружения изменений с использованием объекта ChangeDetectorRef, полученного в конструкторе.

ПРИМЕЧАНИЕ

Метод IterableDifferFactory.create получает необязательный аргумент, определяющий функцию отслеживания. Так директива ngFor реализует условие trackBy (см. главу 13).

Важной частью этого вызова является объект IterableDiffer, назначенный свойству с именем differ, который также может использоваться при вызове метода ngDoCheck.

```
...
ngDoCheck() {
    let changes = this.differ.diff(this.dataSource);
    if (changes != null) {
        console.log("ngDoCheck called, changes detected");
        changes.forEachAddedItem(addition => {
            this.container.createEmbeddedView(this.template,
                new PaIteratorContext(addition.item,
                    addition.currentIndex, changes.length));
        });
    }
}
...
```

Метод IterableDiffer.diff получает объект для сравнения и возвращает список изменений или null при отсутствии изменений. Проверка null позволяет директиве избежать лишней работы, если метод ngDoCheck вызывается из-за изменений в других местах приложения. Объект, возвращаемый методом IterableDiffer.diff, предоставляет свойства и методы обработки изменений, перечисленные в табл. 16.4.

Таблица 16.4. Методы и свойства результата IterableDiffer.Diff

Имя	Описание
collection	Свойство возвращает коллекцию объектов, проанализированных на наличие изменений
length	Свойство возвращает количество объектов в коллекции
forEachItem(func)	Метод вызывает заданную функцию для каждого объекта в коллекции
forEachPreviousItem(func)	Метод вызывает заданную функцию для каждого объекта в предыдущей версии коллекции
forEachAddedItem(func)	Метод вызывает заданную функцию для каждого нового объекта в коллекции

Имя	Описание
forEachMovedItem(func)	Метод вызывает заданную функцию для каждого объекта, позиция которого изменилась
forEachRemovedItem(func)	Метод вызывает заданную функцию для каждого объекта, который был удален из коллекции
forEachIdentityChange(func)	Метод вызывает заданную функцию для каждого объекта с изменившейся идентичностью

Функции, передаваемые методам из табл. 16.4, получают объект `CollectionChangeRecord`, описывающий объект и его изменения при помощи свойств, перечисленных в табл. 16.5.

Таблица 16.5. Свойства CollectionChangeRecord

Имя	Описание
item	Свойство возвращает объект данных
trackById	Свойство возвращает идентификационное значение, если используется функция trackBy
currentIndex	Свойство возвращает текущий индекс объекта в коллекции
previousIndex	Свойство возвращает предыдущий индекс объекта в коллекции

Коду листинга 16.16 достаточно обрабатывать новые объекты в источнике данных, потому что только такое изменение может вноситься остальным кодом приложения. Если результат метода `IterableDiffer.diff` отличен от `null`, то метод `forEachAddedItem` используется для активизации функции `=>` для каждого нового обнаруженного объекта. Функция вызывается по одному разу для каждого нового объекта и использует свойства табл. 16.5 для создания новых представлений в контейнере представлений.

Изменения в листинге 16.16 включают новое консольное сообщение, которое выводится на консоль JavaScript в браузере только в том случае, если директива обнаруживает изменение в данных. Повторив процесс добавления нового товара, вы увидите, что сообщение выводится только при исходном запуске приложения и при нажатии кнопки Create. Метод `ngDoCheck` все равно вызывается, и директиве приходится каждый раз проверять изменения в данных, поэтому избыточная работа все равно выполняется. Тем не менее такие операции менее затратны и занимают меньше времени, чем уничтожение элементов HTML с их повторным созданием.

ПРИМЕЧАНИЕ

Angular также поддерживает отслеживание изменений для пар «ключ — значение», что позволяет отслеживать объекты Map и объекты, использующие свойства как ключи для выборки. Пример приведен в главе 19.

Отслеживание представлений

Обнаружение изменений реализуется просто, если речь идет о создании новых объектов данных. Другие операции, такие как обработка удалений или редактирования, реализуются сложнее, и директиве приходится следить за тем, какое представление связано с тем или иным объектом данных.

Для демонстрации мы добавим поддержку удаления объектов `Product` из модели данных. Сначала в листинге 16.17 в компонент добавляется метод для удаления товара по ключу. Это не является обязательным требованием, потому что шаблон может обращаться к репозиторию через свойство `model` компонента, но оно делает код приложения более понятным, так как все обращения к данным и их использование проходят по одной схеме.

Листинг 16.17. Добавление метода удаления в файл component.ts

```
import { ApplicationRef, Component } from "@angular/core";
import { NgForm } from "@angular/forms";
import { Model } from "./repository.model";
import { Product } from "./product.model";
import { ProductFormGroup } from "./form.model";

@Component({
    selector: "app",
    templateUrl: "app/template.html"
})
export class ProductComponent {
    model: Model = new Model();
    form: ProductFormGroup = new ProductFormGroup();

    getProduct(key: number): Product {
        return this.model.getProduct(key);
    }

    getProducts(): Product[] {
        return this.model.getProducts();
    }

    newProduct: Product = new Product();
    addProduct(p: Product) {
        this.model.saveProduct(p);
    }

    deleteProduct(key: number) {
        this.model.deleteProduct(key);
    }

    formSubmitted: boolean = false;
    submitForm(form: NgForm) {
        this.formSubmitted = true;
        if (form.valid) {
            this.addProduct(this.newProduct);
```

```
            this.newProduct = new Product();
            form.reset();
            this.formSubmitted = false;
        }
    }

    showTable: boolean = true;
}
```

В листинге 16.18 обновляется шаблон: контент, генерируемый структурной директивой, содержит столбец элементов `button`. Каждая кнопка удаляет объект данных, связанный со строкой, в которой она находится.

Листинг 16.18. Добавление кнопки удаления в файле template.html

```
...
<table *paIf="showTable"
    class="table table-sm table-bordered table-striped">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th><th></th></tr>
    <tr *paFor="let item of getProducts(); let i = index; let odd = odd;
            let even = even" [class.bg-info]="odd" [class.bg-warning]="even">
        <td style="vertical-align:middle">{{i + 1}}</td>
        <td style="vertical-align:middle">{{item.name}}</td>
        <td style="vertical-align:middle">{{item.category}}</td>
        <td style="vertical-align:middle">{{item.price}}</td>
        <td class="text-xs-center">
            <button class="btn btn-danger btn-sm" (click)="deleteProduct(item.id)">
                Delete
            </button>
        </td>
    </tr>
</table>
...
```

Элементы `button` содержат привязки события `click`, которые вызывают метод `deleteProduct` компонента. Также для существующих элементов `td` задается значение стилевого свойства CSS `vertical-align`, чтобы текст в таблице выравнивался по тексту кнопки. Последний шаг — обработка изменений данных в структурной директиве и удаление объекта из источника данных (листинг 16.19).

Листинг 16.19. Обработка удаления объекта в файле iterator.directive.ts

```
import { Directive, ViewContainerRef, TemplateRef,
         Input, SimpleChange, IterableDiffer, IterableDiffers,
         ChangeDetectorRef, CollectionChangeRecord, ViewRef } from "@angular/core";

@Directive({
    selector: "[paForOf]"
})
export class PaIteratorDirective {
    private differ: IterableDiffer;
    private views: Map<any, PaIteratorContext> = new Map<any, PaIteratorContext>();
```

```
    constructor(private container: ViewContainerRef,
        private template: TemplateRef<Object>,
        private differs: IterableDiffers,
        private changeDetector: ChangeDetectorRef) {
    }

    @Input("paForOf")
    dataSource: any;

    ngOnInit() {
        this.differ = this.differs.find(this.dataSource).create(this.
changeDetector);
    }

    ngDoCheck() {
        let changes = this.differ.diff(this.dataSource);
        if (changes != null) {
            changes.forEachAddedItem(addition => {
                let context = new PaIteratorContext(addition.item,
                    addition.currentIndex, changes.length);
                context.view = this.container.createEmbeddedView(this.template,
                    context);
                this.views.set(addition.trackById, context);
            });
            let removals = false;
            changes.forEachRemovedItem(removal => {
                removals = true;
                let context = this.views.get(removal.trackById);
                if (context != null) {
                    this.container.remove(this.container.indexOf(context.view));
                    this.views.delete(removal.trackById);
                }
            });
            if (removals) {
                let index = 0;
                this.views.forEach(context =>
                    context.setData(index++, this.views.size));
            }
        }
    }
}

class PaIteratorContext {
    index: number;
    odd: boolean; even: boolean;
    first: boolean; last: boolean;
    view: ViewRef;

    constructor(public $implicit: any,
            public position: number, total: number ) {
        this.setData(position, total);
    }

    setData(index: number, total: number) {
```

```
        this.index = index;
        this.odd = index % 2 == 1;
        this.even = !this.odd;
        this.first = index == 0;
        this.last = index == total - 1;
    }
}
```

Обработка удаления объектов требует двух операций. Первая — обновление набора представлений с удалением тех, которые соответствуют объектам, предоставляемым методом `forEachRemovedItem`. Это подразумевает отслеживание соответствий между объектами данных и представлениями, с которыми они связаны; для этого я добавил свойство `ViewRef` в класс `PaIteratorContext` и использовал `Map` для сбора данных, индексированных по значению свойства `CollectionChangeRecord.trackById`.

РАБОТА С КАРТАМИ

Класс Map, представляющий коллекцию пар «ключ — значение», определяется в спецификации ES6. В TypeScript имеется встроенная поддержка управления парами «ключ — значение» для ключей типа string или number, но в нашем примере она пользы не приносит. При использовании Map можно предоставить аннотации TypeScript для типов ключей и значений:

```
...
private views: Map<any, PaIteratorContext> = new Map<any, PaIteratorContext>();
...
```

В этом примере any означает, что ключом может быть произвольный объект JavaScript, что позволяет использовать идентификаторы для хранения объектов контекста в Map. Пары «ключ — значение» создаются методом set:

```
...
this.views.set(addition.trackById, context);
...
```

Выборка данных осуществляется методом get, а удаление из Map — методом delete.

```
...
let context = this.views.get(removal.trackById);
...
this.views.delete(removal.trackById);
...
```

Содержимое Map обрабатывается с использованием метода forEach, который активизирует функцию обратного вызова для каждой пары «ключ — значение». Как объясняется в главах приложения SportsStore, Map плохо подходит на роль источника данных для привязок данных.

Полное описание функциональности Map доступно по адресу https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map. В старых браузерах встроенная поддержка Map отсутствует; для них приходится использовать библиотеку совместимости, такую как core-js (см. главу 11).

При обработке изменений в коллекциях директива обрабатывает каждый удаленный объект: она читает соответствующий объект `PaIteratorContext` из `Map`, получает его объект `ViewRef` и передает его методу `ViewContainerRef.remove` для удаления контента, связанного с объектом из контейнера представлений.

Вторая операция — обновление данных контекста для оставшихся объектов, чтобы привязки, зависящие от позиции представлений в контейнере представлений, обновлялись правильно. Директива вызывает метод `PaIteratorContext.setData` для каждого объекта контекста, оставшегося в `Map`, для обновления позиции представления в контейнере и общего количества используемых представлений. Без этих изменений свойства, предоставляемые объектом контекста, не будут точно отражать модель данных, а это означает, что цвета строк не будут чередоваться, а кнопки Delete будут относиться к неправильным объектам.

В результате таких изменений в каждой строке таблицы содержится кнопка Delete, которая удаляет соответствующий объект из модели данных, что, в свою очередь, инициирует обновление таблицы (рис. 16.7).

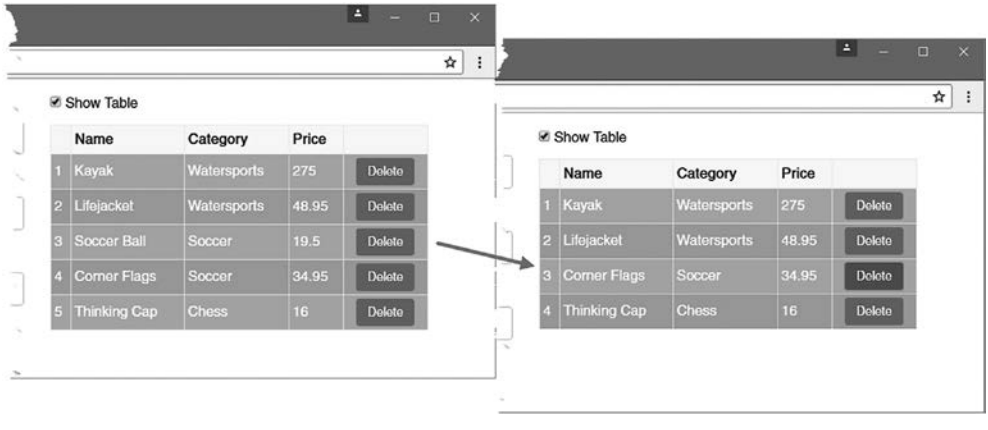


Рис. 16.7. Удаление объектов из модели данных

Запрос контента управляющего элемента

Директивы могут запрашивать контент своего управляющего элемента, чтобы получить доступ к содержащимся в нем директивам (*контентным потомкам*). Это позволяет директивам координировать совместную работу.

ПРИМЕЧАНИЕ

Также возможна совместная работа директив, основанная на общем доступе к службам (см. главу 19).

Чтобы показать, как происходит запрос контента, создайте файл cellColor.directive.ts в папке app и включите в него определение директивы из листинга 16.20.

Листинг 16.20. Содержимое файла cellColor.directive.ts в папке app

```
import { Directive, HostBinding } from "@angular/core";

@Directive({
    selector: "td"
})
export class PaCellColor {

    @HostBinding("class")
    bgClass: string = "";

    setColor(dark: Boolean) {
        this.bgClass = dark ? "bg-inverse" : "";
    }
}
```

Класс `PaCellColor` определяет простую директиву атрибута, которая работает с элементами `td` и привязывается к свойству `class` управляющего элемента. Метод `setColor` получает логический параметр, который при значении `true` задает свойству `class` значение `bg-inverse` (класс Bootstrap для инвертированных цветов — темный фон со светлым текстом).

Класс `PaCellColor` станет директивой, которая внедряется в контент управляющего элемента в этом примере. Наша цель — написать другую директиву, которая будет обращаться к управляющему элементу с запросом на поиск встроенной директивы для вызова его метода `setColor`. Для этого создайте файл cellColorSwitcher.directive.ts в папке app и включите в него определение директивы из листинга 16.21.

Листинг 16.21. Содержимое файла cellColorSwitcher.directive.ts в папке app

```
import { Directive, Input, Output, EventEmitter,
         SimpleChange, ContentChild } from "@angular/core";
import { PaCellColor } from "./cellColor.directive";

@Directive({
    selector: "table"
})
export class PaCellColorSwitcher {

    @Input("paCellDarkColor")
    modelProperty: Boolean;

    @ContentChild(PaCellColor)
    contentChild: PaCellColor;

    ngOnChanges(changes: { [property: string]: SimpleChange }) {
        if (this.contentChild != null) {
            this.contentChild.setColor(changes["modelProperty"].currentValue);
        }
    }
}
```

Класс `PaCellColorSwitcher` определяет директиву, которая работает с элементами `table` и определяет входное свойство с именем `paCellDarkColor`. Важной частью этой директивы является свойство `contentChild`.

```
...
@ContentChild(PaCellColor)
contentChild: PaCellColor;
...
```

Декоратор `@ContentChild` сообщает Angular, что директива должна запросить контент управляющего элемента и присвоить первый результат запроса свойству. В аргументе декоратора `@ContentChild` передается один или несколько классов директив. В данном примере аргумент декоратора `@ContentChild` содержит `PaCellColor`; тем самым мы приказываем Angular найти первый объект `PaCellColor` в контенте управляющего элемента и задать его свойству, помеченному декоратором.

ПРИМЕЧАНИЕ

В запросах также могут использоваться имена переменных шаблонов; так, конструкция @ContentChild("myVariable") найдет первую директиву, присвоенную myVariable.

Результат запроса предоставляет директиве `PaCellColorSwitcher` доступ к дочернему компоненту и разрешает ей вызвать метод `setColor` в ответ на изменения во входном свойстве.

ПРИМЕЧАНИЕ

Если вы захотите включить в результаты потомков дочерних директив, настройте запрос соответствующим образом — например, @ContentChild(PaCellColor, { descendants: true}).

В листинге 16.22 в шаблон добавляется флажок, использующий директиву `ngModel` для присваивания значения переменной, привязанной к входному свойству директивы `PaCellColorSwitcher`.

Листинг 16.22. Применение директив в файле template.html

```
...
<div class="col-xs-8">
    <div class="checkbox">
        <label>
            <input type="checkbox" [(ngModel)]="showTable" />
            Show Table
        </label>
    </div>

    <div class="checkbox">
        <label>
            <input type="checkbox" [(ngModel)]="darkColor" />
            Dark Cell Color
        </label>
    </div>
```

```
    <table *paIf="showTable" [paCellDarkColor]="darkColor"
            class="table table-sm table-bordered table-striped">
        <tr><th></th><th>Name</th><th>Category</th><th>Price</th><th></th></tr>
        <tr *paFor="let item of getProducts(); let i = index; let odd = odd;
                let even = even" [class.bg-info]="odd" [class.bg-warning]="even">
            <td style="vertical-align:middle">{{i + 1}}</td>
            <td style="vertical-align:middle">{{item.name}}</td>
            <td style="vertical-align:middle">{{item.category}}</td>
            <td style="vertical-align:middle">{{item.price}}</td>
            <td class="text-xs-center">
                <button class="btn btn-danger btn-sm" (click)="deleteProduct(i)">
                    Delete
                </button>
            </td>
        </tr>
    </table>
</div>
...
```

Остается сделать последний шаг — зарегистрировать новые директивы в свойстве `declarations` (листинг 16.23).

Листинг 16.23. Регистрация новых директив в файле app.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { ProductComponent } from "./component";
import { FormsModule, ReactiveFormsModule  } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
import { PaModel } from "./twoway.directive";
import { PaStructureDirective } from "./structure.directive";
import { PaIteratorDirective } from "./iterator.directive";
import { PaCellColor } from "./cellColor.directive";
import { PaCellColorSwitcher } from "./cellColorSwitcher.directive";

@NgModule({
    imports: [BrowserModule, FormsModule, ReactiveFormsModule],
    declarations: [ProductComponent, PaAttrDirective, PaModel,
        PaStructureDirective, PaIteratorDirective,
        PaCellColor, PaCellColorSwitcher],
    bootstrap: [ProductComponent]
})
export class AppModule { }
```

После сохранения изменений над таблицей появляется новый флажок. Если установить его, директива `ngModel` инициирует обновление входного свойства директивы `PaCellColorSwitcher`. Это приводит к вызову метода `setColor` объекта директивы `PaCellColor`, найденного с использованием декоратора `@ContentChild`. Визуальный эффект незначителен: изменения распространяются только на первую директиву `PaCellColor` для ячейки с цифрой 1 в левом верхнем углу таблицы (рис. 16.8).

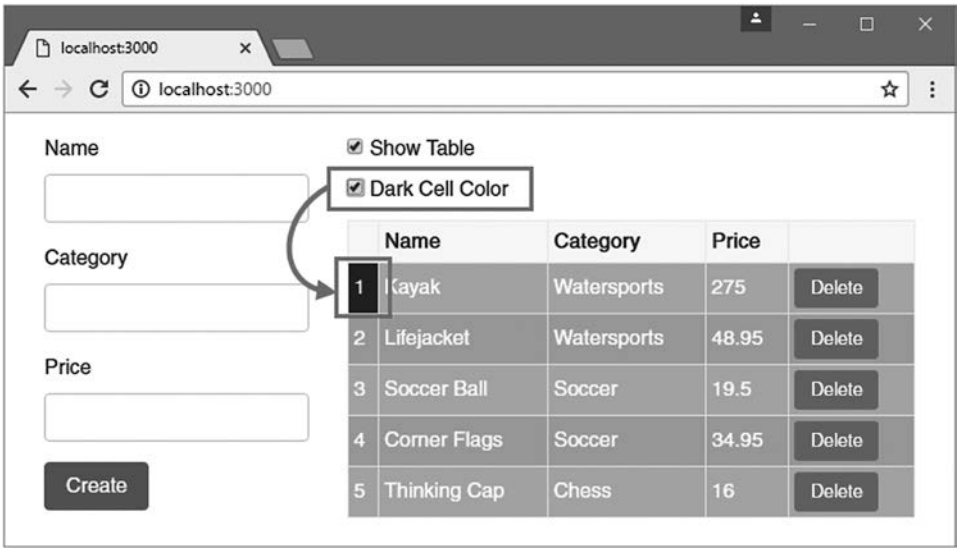


Рис. 16.8. Работа с контентным потомком

Получение информации о нескольких контентных потомках

Декоратор `@ContentChild` находит первый объект директивы, соответствующий аргументу, и задает его свойству `decorated`. Если вы хотите получить все объекты директив, соответствующие аргументу, используйте декоратор `@ContentChildren` (листинг 16.24).

Листинг 16.24. Получение информации о нескольких контентных потомках в файле cellColorSwitcher.directive.ts

```
import { Directive, Input, Output, EventEmitter,
         SimpleChange, ContentChildren, QueryList } from "@angular/core";
import { PaCellColor } from "./cellColor.directive";

@Directive({
    selector: "table"
})
export class PaCellColorSwitcher {

    @Input("paCellDarkColor")
    modelProperty: Boolean;

    @ContentChildren(PaCellColor)
    contentChildren: QueryList<PaCellColor>;

    ngOnChanges(changes: { [property: string]: SimpleChange }) {
        this.updateContentChildren(changes["modelProperty"].currentValue);
    }

    private updateContentChildren(dark: Boolean) {
        if (this.contentChildren != null && dark != undefined) {
```

```
            this.contentChildren.forEach((child, index) => {
                child.setColor(index % 2 ? dark : !dark);
            });
        }
    }
}
```

При использовании декоратора `@ContentChildren` результаты запроса предоставляются через объект `QueryList`, который дает доступ к объектам директив с использованием методов и свойств, перечисленных в табл. 16.6.

Таблица 16.6. Свойства и методы класса QueryList

Имя	Описание
length	Свойство возвращает количество найденных объектов директив
first	Свойство возвращает первый найденный объект директивы
last	Свойство возвращает последний найденный объект директивы
map(function)	Метод вызывает функцию для каждого найденного объекта директивы и создает новый массив (эквивалент метода Array.map)
filter(function)	Метод вызывает функцию для каждого найденного объекта директивы и создает массив с объектами, для которых функция вернула true (эквивалент метода Array.filter)
reduce(function)	Метод вызывает функцию для каждого найденного объекта директивы и вычисляет одно значение (эквивалент метода Array.reduce)
forEach(function)	Метод вызывает функцию для каждого найденного объекта директивы (эквивалент метода Array.forEach)
some(function)	Метод вызывает функцию для каждого найденного объекта директивы и возвращает true, если функция вернула true хотя бы для одного объекта (эквивалент метода Array.some)
changes	Свойство используется для отслеживания результатов изменений (см. раздел «Получение уведомлений об изменениях в запросах»)

В листинге директива реагирует на изменения входного свойства, вызывая метод `updateContentChildren`, который, в свою очередь, использует метод `forEach` для `QueryList` и вызывает метод `setColor` для каждой второй директивы, соответствующей запросу. На рис. 16.9 показан эффект установки флажка.

Получение уведомлений об изменениях в запросах

Результаты запросов контента являются «живыми»; это означает, что они автоматически обновляются в соответствии с добавлением, изменением или удалением контента управляющего элемента. Для получения уведомлений об изменениях в результатах запросов следует использовать интерфейс `Observable`, который предоставляется пакетом Reactive Extensions, добавленным в проект в главе 11. Пакет Reactive Extensions предоставляет механизм распространения событий изменений в приложениях JavaScript. О том, как работают объекты `Observable`, более подробно рассказано в главе 23, а пока достаточно сказать, что они используются Angular для управления изменениями.

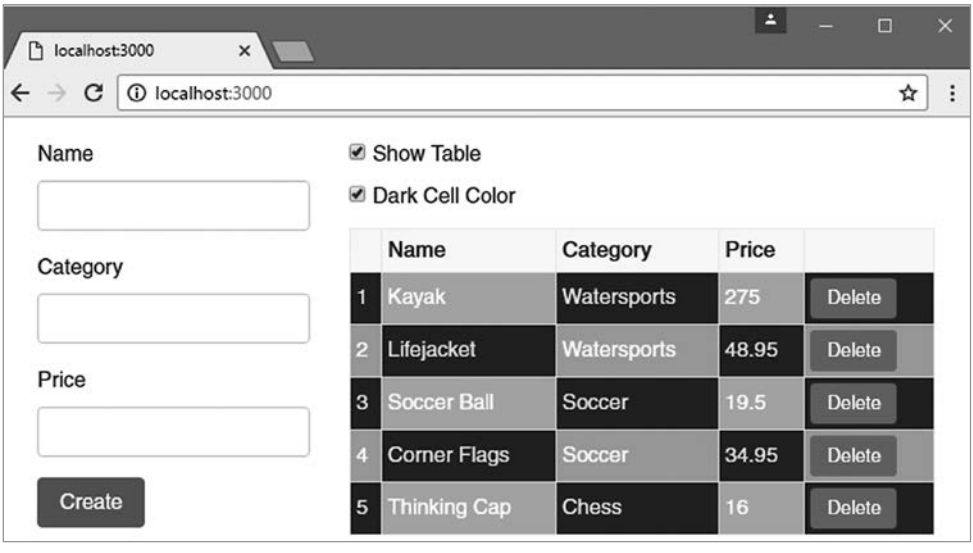


Рис. 16.9. Работа с несколькими контентными потомками

В листинге 16.25 директива `PaCellColorSwitcher` обновляется для получения уведомлений при изменении набора контентных потомков в `QueryList`.

Листинг 16.25. Получение уведомлений об изменениях в файле cellColorSwitcher.directive.ts

```
import { Directive, Input, Output, EventEmitter,
         SimpleChange, ContentChildren, QueryList } from "@angular/core";
import { PaCellColor } from "./cellColor.directive";

@Directive({
    selector: "table"
})
export class PaCellColorSwitcher {

    @Input("paCellDarkColor")
    modelProperty: Boolean;

    @ContentChildren(PaCellColor)
    contentChildren: QueryList<PaCellColor>;

    ngOnChanges(changes: { [property: string]: SimpleChange }) {
        this.updateContentChildren(changes["modelProperty"].currentValue);
    }

    ngAfterContentInit() {
        this.contentChildren.changes.subscribe(() => {
            setTimeout(() => this.updateContentChildren(this.modelProperty), 0);
        });
    }

    private updateContentChildren(dark: Boolean) {
        if (this.contentChildren != null && dark != undefined) {
            this.contentChildren.forEach((child, index) => {
```

```
                child.setColor(index % 2 ? dark : !dark);
            });
        }
    }
}
```

Значение свойства запроса контентных потомков задается только после вызова метода жизненного цикла `ngAfterContentInit`, поэтому я использую этот метод для организации уведомлений. Класс `QueryList` определяет метод `changes`, который возвращает объект Reactive Extensions `Observable`, определяющий метод `subscribe`. Этот метод получает функцию, которая вызывается при изменении содержимого `QueryList`, другими словами, при изменении набора директив, определяемых аргументом декоратора `@ContentChildren`. Функция, которая передается методу `subscribe`, вызывает метод `updateContentChildren` для задания цветов, но использует для этого вызов функции `setTimeout`, откладывающий выполнение вызова метода до завершения функции обратного вызова `subscribe`. Без вызова `setTimeout` Angular сообщит об ошибке, потому что директива пытается начать новое обновление контента до завершения полной обработки текущего обновления. В результате таких изменений всем новым ячейкам таблицы, создаваемым на форме HTML, автоматически назначается темный цвет (рис. 16.10).

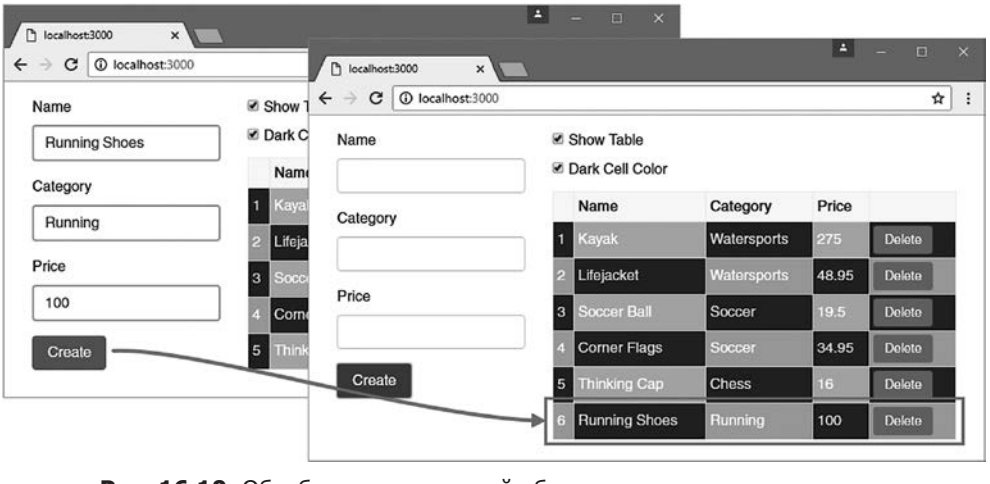


Рис. 16.10. Обработка уведомлений об изменении контента запросов

Итоги

В этой главе я объяснил, как работают структурные директивы, на примере воссоздания функциональности встроенных директив `ngIf` и `ngFor`. Мы рассмотрели использование контейнеров представлений и шаблонов, полный и компактный синтаксис применения структурных директив, изучили тему создания директив для перебора коллекции объектов данных и возможности получения информации о контенте управляющих элементов в директивах. В следующей главе будут рассмотрены компоненты и их отличия от директив.

17 Компоненты

Компоненты представляют собой директивы, которые обладают собственными шаблонами (вместо использования контента, предоставляемого другим источником). Компонентам доступна вся функциональность директив, описанная в предыдущих главах; они также имеют управляющий элемент, могут определять входные и выходные свойства и т. д. Но при этом они также определяют собственный контент.

Важность шаблонов не сразу понятна, но у структурных директив и директив атрибутов есть свои ограничения. Директивы могут выполнять полезную работу, но они мало что знают об элементах, к которым они применяются. Директивы наиболее эффективно работают как инструменты общего назначения — например, директива `ngModel` может применяться к любому свойству модели данных и к любому элементу формы независимо от того, для чего используются эти данные или элемент.

С другой стороны, компоненты тесно привязаны к содержимому своих шаблонов. Компоненты предоставляют данные и логику, которые будут использоваться привязками данных, применяемыми элементами HTML в шаблоне; шаблон предоставляет контекст, используемый для вычисления выражений привязки данных, и служит промежуточным звеном между директивами и остальными частями приложения. Компоненты также помогают разбить крупные проекты Angular на несколько более удобных фрагментов.

В этой главе вы узнаете, как работают компоненты и как изменить структуру приложения с введением дополнительных компонентов. В табл. 17.1 компоненты представлены в контексте.

Таблица 17.1. Директивы атрибутов в контексте

Вопрос	Ответ
Что это такое?	Компоненты — директивы, определяющие собственный контент HTML (а возможно, и собственные стили CSS)
Для чего они нужны?	Компоненты позволяют определять автономные функциональные блоки, которые упрощают управление проектом и повторное использование функциональности
Как они используются?	Декоратор @Component применяется к классу, зарегистрированному в модуле Angular приложения

Вопрос	Ответ
Есть ли у них недостатки или скрытые проблемы?	Нет. Компоненты поддерживают всю функциональность директив, дополняя ее возможностью определения собственных шаблонов
Есть ли альтернативы?	Приложение Angular должно содержать как минимум один компонент, используемый в процессе начальной загрузки. В остальном вы не обязаны определять дополнительные компоненты, хотя приложение может стать громоздким, а у разработчика могут возникнуть сложности с управлением кодом

В табл. 17.2 приведена краткая сводка материала главы.

Таблица 17.2. Сводка материала главы

Проблема	Решение	Листинг
Создание компонента	Примените декоратор @Component к классу	1–5
Определение контента, отображаемого компонентом	Создайте встроенный или внешний шаблон	6–8
Включение данных в шаблон	Используйте привязку данных в шаблоне компонента	9
Координация работы компонентов	Используйте входные или выходные свойства	10–16
Вывод контента из элемента, к которому применяется компонент	Спроецируйте контент управляющего элемента	17–21
Стилевое оформление контента	Создайте стили компонента	22–30
Запрос контента в шаблоне компонента	Используйте декоратор @ViewChildren	31

Подготовка проекта

В этой главе мы продолжим использовать проект example, который был создан в главе 11 и дорабатывался в последующих главах. Никакие дополнительные изменения при подготовке к этой главе не потребуются.

ПРИМЕЧАНИЕ

Проект также можно взять из архива исходного кода, прилагаемого к книге. Его можно бесплатно загрузить на сайте издательства apress.com.

Чтобы запустить компилятор TypeScript и сервер HTTP для разработки, выполните следующую команду из папки example:

```
npm start
```

Сервер HTTP для разработки открывает новое окно браузера (рис. 17.1).

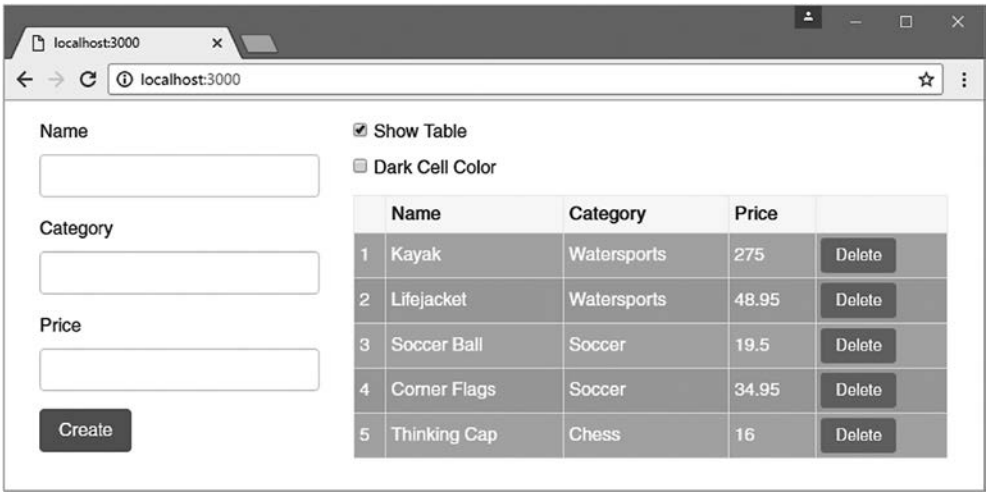


Рис. 17.1. Запуск приложения

Применение компонентов для формирования структуры приложения

На данный момент проект содержит всего один компонент и один шаблон. Приложениям Angular требуется как минимум один компонент, называемый *корневым компонентом*; он определяет точку входа, задаваемую в модуле Angular.

Проблема, возникающая при наличии только одного компонента, заключается в том, что в конечном итоге он содержит всю логику, необходимую для всей функциональности приложения, а его шаблон содержит всю разметку, обеспечивающую доступ к этой функциональности. В результате один компонент и его шаблон имеют множество обязанностей. Компонент в нашем примере:

- предоставляет Angular точку входа приложения (как корневой компонент);
- предоставляет доступ к модели данных приложения для использования в привязках данных;
- определяет форму HTML для создания новых товаров;
- определяет таблицу HTML для вывода информации о товарах;
- определяет макет, содержащий форму и таблицу;
- проверяет данные формы при создании нового товара;
- поддерживает информацию состояния, необходимую для предотвращения использования недействительных данных при создании товара;

- поддерживает информацию состояния о том, должна ли таблица отображаться в приложении.

Даже для такого простого приложения этого слишком много, а большая часть этих задач никак не связана между собой. Этот эффект постепенно формируется в процессе разработки, но в конечном итоге он усложняет тестирование приложений, так как отдельные функции не удается эффективно изолировать друг от друга. Кроме того, постепенное усложнение кода и разметки затрудняет сопровождение и доработку приложения.

Добавление новых компонентов в приложение позволяет разделить его функциональность на структурные блоки, которые можно заново использовать в разных частях приложения. Далее мы создадим компоненты, которые разбивают функциональность нашего примера на удобные, пригодные для повторного использования и автономные блоки. Попутно будут рассмотрены возможности, которые предоставляют компоненты в дополнение к тем, которые предоставляются директивами. Подготовка к этим изменениям начинается с упрощения шаблона существующего компонента (листинг 17.1).

Листинг 17.1. Упрощение контента в файле template.html

```
<div class="col-xs-4 p-a-1 bg-success">
    Form will go here
</div>
<div class="col-xs-8 p-a-1 bg-primary">
    Table will go here
</div>
```

Сохранив изменения в шаблоне, вы увидите контент, показанный на рис. 17.2. Временные заполнители будут заменяться функциональностью приложения по мере разработки новых компонентов и включения их в приложение.

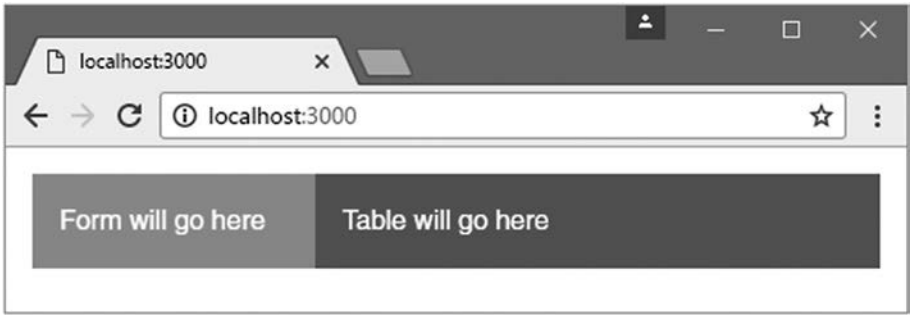


Рис. 17.2. Упрощение существующего шаблона

Создание новых компонентов

Чтобы создать новый компонент, создайте файл productTable.component.ts в папке app и включите в него определение компонента из листинга 17.2.

Листинг 17.2. Содержимое файла productTable.component.ts в папке app

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

@Component({
    selector: "paProductTable",
    template: "<div>This is the table component</div>"
})
export class ProductTableComponent {

}
```

Компонент представляет собой класс, к которому был применен декоратор `@Component`. Это минимальный уровень требований, при котором компонент обладает достаточной функциональностью, чтобы считаться компонентом, но при этом еще не делает ничего полезного.

Файлам, определяющим компоненты, принято присваивать содержательное имя, описывающее предназначение компонента, а затем через точку добавлять суффикс component.ts. Для компонента из нашего примера, который будет использоваться для генерирования таблицы товаров, имя файла имеет вид productTable.component.ts. Имя класса компонента (`ProductTableComponent`) также должно быть содержательным.

Декоратор `@Component` описывает и настраивает компонент. Самые полезные свойства декоратора перечислены в табл. 17.3, вместе со ссылками для получения дополнительной информации (не все свойства рассматриваются в этой главе).

Таблица 17.3. Свойства декоратора компонента

Имя	Описание
animations	Свойство используется для настройки анимаций (см. главу 28)
encapsulation	Свойство используется для изменения настроек инкапсуляции представления, управляющих изоляцией стилей компонента от остальной части документа HTML. За дополнительной информацией обращайтесь к разделу «Настройка инкапсуляции представления»
moduleId	Свойство задает модуль, в котором определяется компонент, и используется в сочетании со свойством templateUrl (см. главу 12)
selector	Свойство задает селектор CSS, используемый для выбора управляющих элементов (см. объяснение после таблицы)
styles	Свойство используется для определения стилей CSS, применяемых только к шаблону компонента. Стили определяются как встроенные в файле TypeScript. За дополнительной информацией обращайтесь к разделу «Использование стилей компонентов»
styleUrls	Свойство используется для определения стилей CSS, применяемых только к шаблону компонента. Стили определяются в отдельных файлах CSS. За дополнительной информацией обращайтесь к разделу «Использование стилей компонентов»

Имя	Описание
template	Свойство используется для задания встроенного шаблона (см. раздел «Определение шаблонов»)
templateUrl	Свойство используется для задания внешнего шаблона (см. раздел «Определение шаблонов»)
providers	Свойство используется для создания локальных провайдеров для служб (см. главу 19)
viewProviders	Свойство используется для создания локальных провайдеров для служб, доступных только для потомков представлений (см. главу 20)

Для второго компонента создайте файл productForm.component.ts в папке app и включите в него код из листинга 17.3.

Листинг 17.3. Содержимое файла productForm.component.ts в папке app

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

@Component({
    selector: "paProductForm",
    template: "<div>This is the form component</div>"
})
export class ProductFormComponent {

}
```

Этот компонент не сложнее предыдущего; пока он всего лишь резервирует место в макете. Позже в этой главе мы добавим в него более полезную функциональность. Чтобы компоненты могли использоваться в приложении, их необходимо объявить в модуле Angular приложения (листинг 17.4).

Листинг 17.4. Объявление новых компонентов в файле app.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { ProductComponent } from "./component";
import { FormsModule, ReactiveFormsModule  } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
import { PaModel } from "./twoway.directive";
import { PaStructureDirective } from "./structure.directive";
import { PaIteratorDirective } from "./iterator.directive";
import { PaCellColor } from "./cellColor.directive";
import { PaCellColorSwitcher } from "./cellColorSwitcher.directive";
import { ProductTableComponent } from "./productTable.component";
import { ProductFormComponent } from "./productForm.component";

@NgModule({
    imports: [BrowserModule, FormsModule, ReactiveFormsModule],
    declarations: [ProductComponent, PaAttrDirective, PaModel,
        PaStructureDirective, PaIteratorDirective,
        PaCellColor, PaCellColorSwitcher, ProductTableComponent,
```

```
        ProductFormComponent],
    bootstrap: [ProductComponent]
})
export class AppModule { }
```

Класс компонента подключается командой `import` и добавляется в массив `declarations` декоратора `NgModule`.

Остается добавить элемент HTML, соответствующий свойству `selector` компонента (листинг 17.5), чтобы предоставить компоненту его управляющий элемент.

Листинг 17.5. Добавление управляющего элемента в файл template.html

```
<div class="col-xs-4 p-a-1 bg-success">
    <paProductForm></paProductForm>
</div>
<div class="col-xs-8 p-a-1 bg-primary">
    <paProductTable></paProductTable>
</div>
```

После того как все изменения будут сохранены, в браузере отображается контент, показанный на рис. 17.3; он показывает, что части документа HTML теперь находятся под управлением новых компонентов.

Рис. 17.3. Добавление новых компонентов

Новая структура приложения

Новые компоненты изменили структуру приложения. Ранее корневой компонент отвечал за весь контент HTML, выводимый приложением. Теперь приложение содержит три компонента, и ответственность за часть контента HTML была передана новым компонентам (рис. 17.4).

Когда браузер загружает файл index.html, запускается процесс начальной загрузки Angular. Angular обрабатывает модуль приложения, в котором содержится список компонентов, необходимых приложению. Angular проверяет декоратор каждого компонента в своей конфигурации, включая значение свойства `selector`, которое используется для идентификации управляющих элементов.

Затем Angular начинает обработку тела файла index.html и находит элемент app, заданный свойством `selector` компонента `ProductComponent`. Angular заполняет элемент `app` шаблоном компонента, содержащимся в файле template.html. Angular

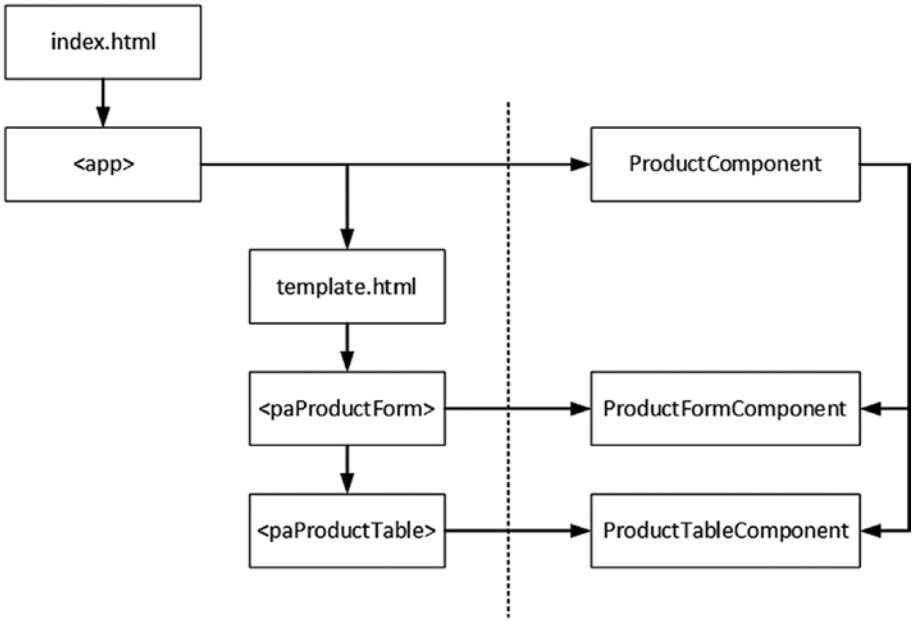


Рис. 17.4. Новая структура приложения

анализирует содержимое файла template.html и находит элементы `paProductForm` и `paProductTable`, подходящие по свойствам `selector` вновь добавленных компонентов. Angular заполняет эти элементы шаблонами каждого компонента, создавая временный контент (см. рис. 17.3).

Обратите внимание на ряд важных логических связей. Во-первых, контент HTML, отображаемый в окне браузера, теперь состоит из нескольких шаблонов, каждый из которых находится под управлением компонента. Во-вторых, компонент `ProductComponent` стал родительским по отношению к объектам `ProductFormComponent` и `ProductTableComponent` — эта связь обусловлена тем фактом, что управляющие элементы новых компонентов определяются в файле template.html, содержащем шаблон `ProductComponent` (при этом новые компоненты являются *дочерними* по отношению к `ProductComponent`). Иерархия играет важную роль в области компонентов Angular; вы убедитесь в этом позже, когда речь пойдет о работе компонентов.

Определение шаблонов

Хотя в приложении появились новые компоненты, они пока не оказывают особого влияния, потому что в них отображается только временный контент. Каждый компонент имеет собственный шаблон с определением контента, который будет использоваться для замены управляющего элемента в документе HTML. Шаблоны могут определяться двумя разными способами: как *встроенные* в декораторе `@Component` или как *внешние* в файле HTML.

Чтобы добавить шаблоны для новых компонентов, я присваиваю фрагмент разметки HTML свойству `template` декоратора `@Component`:

```
...
template: "<div>This is the form component</div>"
...
```

Преимуществом такого подхода является простота: компонент и шаблон определяются в одном файле и связь между ними остается однозначной. Недостаток встроенных шаблонов — то, что они могут выйти из-под контроля и начинают плохо читаться, если содержат более нескольких элементов HTML.

Есть и другая проблема: редакторы, выделяющие синтаксические ошибки при вводе, обычно определяют тип выполняемых проверок по расширению файла. Они не понимают, что значение свойства `template` содержит разметку HTML, и рассматривают его как обычную строку.

Если вы используете TypeScript, встроенные шаблоны можно сделать более удобочитаемыми при помощи многострочных строковых литералов. Они заключаются в обратные апострофы (символ `` ` ``) и могут занимать несколько физических строк (листинг 17.6).

Листинг 17.6. Использование многострочных строковых литералов для записи встроенных шаблонов в файле productTable.component.ts

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

@Component({
    selector: "paProductTable",
    template: `<div class='bg-info p-a-1'>
                    This is a multiline template
               </div>`
})
export class ProductTableComponent {

}
```

Многострочные строковые литералы позволяют сохранить структуру элементов HTML в шаблоне. Это упрощает чтение разметки и повышает реальный размер шаблона, который можно включить во встроенном формате, прежде чем он станет слишком громоздким и неудобным. На рис. 17.5 показан эффект применения шаблона из листинга 17.6.

ПРИМЕЧАНИЕ

Я рекомендую использовать внешние шаблоны (см. следующий раздел) для любых шаблонов, содержащих более двух или трех простых элементов, — в основном для того, чтобы использовать преимущества средств редактирования HTML и выделения синтаксиса в современных редакторах. С этими средствами существенно снижается количество ошибок, которые вы обнаружите при запуске своего приложения.

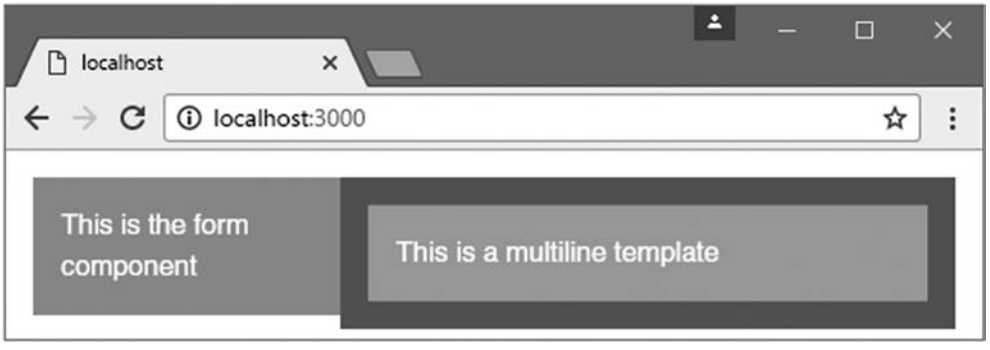


Рис. 17.5. Использование многострочного строкового литерала

Определение внешних шаблонов

Внешние шаблоны определяются в другом файле, отдельно от остальных частей компонента. Преимущество такого подхода заключается в том, что код не смешивается с разметкой HTML, что упрощает чтение и модульное тестирование. Кроме того, редакторы кода будут знать, что при работе с файлом шаблона они имеют дело с контентом HTML; подсветка ошибок сократит их количество на стадии программирования.

К недостаткам внешних шаблонов можно отнести необходимость управления большим количеством файлов в проекте и связывания компонентов с правильными файлами шаблонов. Лучше всего для этого следовать однозначной схеме формирования имен, чтобы по имени файла было сразу видно, что файл содержит шаблон для конкретного компонента. По действующим соглашениям, Angular создает пары файлов по схеме `<`*имя_компонента*`>.component.<`*тип*`>`. Обнаруживая файл productTable.component.ts, вы знаете, что он содержит компонент с именем `Products`, написанный на TypeScript, а если файлу присвоено имя productTable.component.html, он содержит внешний шаблон для компонента `Products`.

ПРИМЕЧАНИЕ

Два типа шаблонов не отличаются по синтаксису и функциональности. Отличается только место хранения компонента (в одном файле с кодом компонента или в отдельном файле).

Чтобы определить внешний шаблон с использованием этой схемы формирования имен, создайте файл productTable.component.html в папке app и включите в него разметку из листинга 17.7.

Листинг 17.7. Содержимое файла productTable.component.html в папке app

```
<div class="bg-info p-a-1">
    This is an external template
</div>
```

Этот шаблон использовался для корневого компонента из главы 11. Для назначения внешнего шаблона свойство `templateURL` используется в декораторе `@Component` (листинг 17.8).

Листинг 17.8. Использование внешнего шаблона в файле productTable.component.html

```
import { Component } from "@angular/core";
@Component({
    selector: "paProductTable",
    templateUrl: "app/productTable.component.html"
})
export class ProductTableComponent {

}
```

Обратите внимание на использование разных свойств: `template` для встроенных шаблонов, `templateUrl` для внешних шаблонов. На рис. 17.6 показан эффект использования внешнего шаблона.

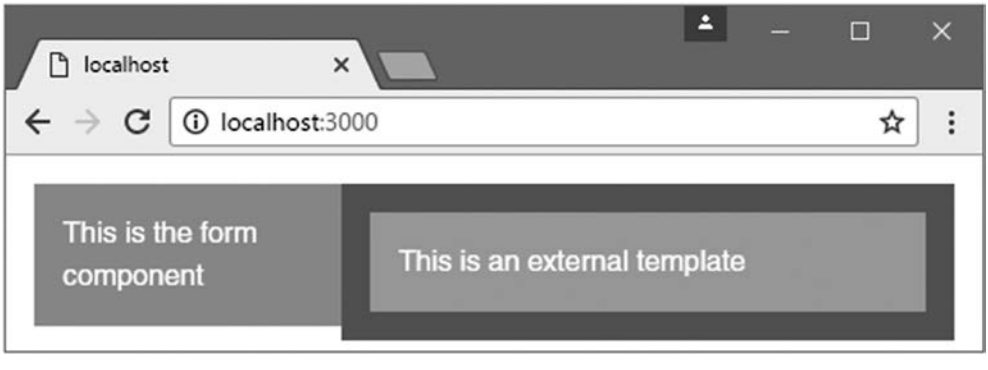


Рис. 17.6. Использование внешнего шаблона

Использование привязок данных в шаблонах компонентов

Шаблон компонента может содержать полный спектр привязок данных, а его целью может быть любая встроенная или нестандартная директива, зарегистрированная в модуле Angular приложения. Каждый класс компонента предоставляет контент для вычисления выражений привязки данных в своем шаблоне, и по умолчанию каждый компонент изолируется от других. Это означает, что компоненту не нужно беспокоиться о возможных конфликтах имен свойств и методов с другими компонентами; он может быть уверен в том, что Angular обеспечит изоляцию. Например, в листинге 17.9 продемонстрировано добавление свойства `model` в дочерний компонент формы, которое бы конфликтовало с одноименным свойством корневого компонента, если бы не изоляция.

Листинг 17.9. Добавление свойства и выражения привязки модели в файле productForm.component.ts

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

@Component({
    selector: "paProductForm",
    template: "<div>{{model}}</div>"
})
export class ProductFormComponent {

    model: string = "This is the model";
}
```

Класс компонента использует свойство `model` для хранения сообщения, отображаемого в шаблоне с использованием привязки со строковой интерполяцией. Результат показан на рис. 17.7.

Рис. 17.7. Использование привязки данных в дочернем компоненте

Использование входных свойств для координации между компонентами

Компоненты редко существуют в полной изоляции — обычно им нужно обмениваться данными с другими частями приложения. Компоненты могут определять входные свойства для получения значений выражений привязки данных со своими управляющими элементами. Выражение будет вычисляться в контексте родительского компонента, но результат будет передаваться свойству дочернего компонента.

Для демонстрации листинг 17.10 добавляет в компонент таблицы входное свойство, которое будет использоваться для получения модели данных для отображения.

Листинг 17.10. Определение входного свойства в файле productTable.component.ts

```
import { Component, Input } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";

@Component({
```

```
    selector: "paProductTable",
    templateUrl: "app/productTable.component.html"
})
export class ProductTableComponent {

    @Input("model")
    dataModel: Model;

    getProduct(key: number): Product {
        return this.dataModel.getProduct(key);
    }

    getProducts(): Product[] {
        return this.dataModel.getProducts();
    }

    deleteProduct(key: number) {
        this.dataModel.deleteProduct(key);
    }

    showTable: boolean = true;
}
```

Теперь компонент определяет входное свойство, которому присваивается значение, присвоенное атрибуту `model` управляющего элемента. Методы `getProduct`, `getProducts` и `deleteProduct` используют входное свойство для обеспечения доступа к модели данных в привязках шаблона компонента (листинг 17.11). Свойство `showTable` будет использоваться при последующей доработке шаблона в листинге 17.14 этой главы.

Листинг 17.11. Добавление привязки данных в файле productTable.component.html

```
There are {{getProducts().length}} items in the model
```

Чтобы предоставить дочернему компоненту необходимые ему данные, следует добавить привязку в его управляющий элемент, определяемый в шаблоне родительского компонента (листинг 17.12).

Листинг 17.12. Добавление привязки данных в файле template.html

```
<div class="col-xs-4 p-a-1 bg-success">
    <paProductForm></paProductForm>
</div>
<div class="col-xs-8 p-a-1 bg-primary">
    <paProductTable [model]="model"></paProductTable>
</div>
```

Эта привязка открывает дочернему компоненту доступ к свойству `model` родительского компонента. Данная возможность может породить путаницу, потому что она зависит от того факта, что управляющий элемент определяется в шаблоне родительского компонента, но входное свойство определяется дочерним компонентом (рис. 17.8).

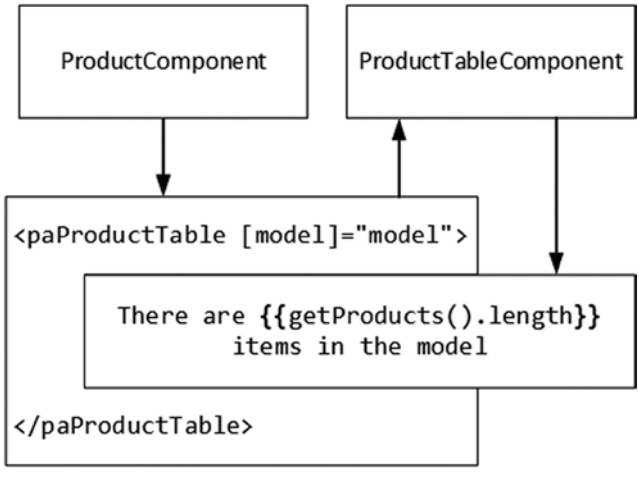


Рис. 17.8. Организация общего доступа к данным между родительским и дочерним компонентами

Управляющий элемент дочернего компонента служит мостом между родительским и дочерним компонентами, а входное свойство позволяет родительскому компоненту передать дочернему компоненту необходимые данные. Результат показан на рис. 17.9.

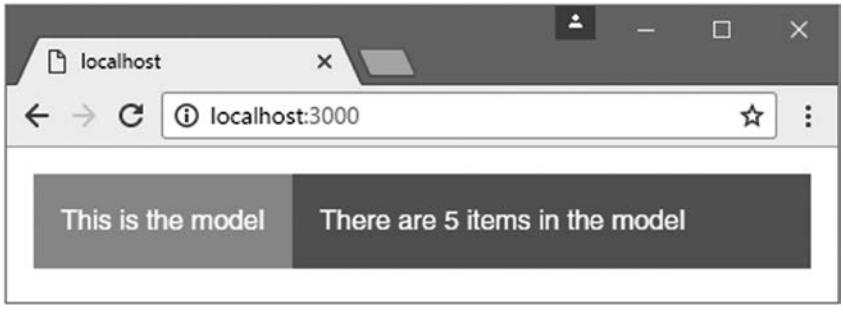


Рис. 17.9. Передача данных от родительского компонента дочернему

Использование директив в шаблоне дочернего компонента

После определения входного свойства дочерний компонент может использовать весь спектр привязок данных и директив — либо с данными, предоставленными родительским компонентом, либо с определением собственных данных. В листинге 17.13 восстанавливается функциональность таблицы из предыдущих глав: на странице выводится список объектов `Product` из модели данных, а также флажок, который определяет, должна ли отображаться таблица. Ранее эта функциональность находилась под управлением корневого компонента и его шаблона.

Листинг 17.13. Восстановление функциональности таблицы в файле productTable.component.html

```
<table class="table table-sm table-bordered table-striped">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th><th></th></tr>
    <tr *paFor="let item of getProducts(); let i = index; let odd = odd;
            let even = even" [class.bg-info]="odd" [class.bg-warning]="even">
        <td style="vertical-align:middle">{{i + 1}}</td>
        <td style="vertical-align:middle">{{item.name}}</td>
        <td style="vertical-align:middle">{{item.category}}</td>
        <td style="vertical-align:middle">{{item.price}}</td>
        <td class="text-xs-center">
            <button class="btn btn-danger btn-sm" (click)="deleteProduct(item.id)">
                Delete
            </button>
        </td>
    </tr>
</table>
```

В разметке используются те же элементы HTML, привязки данных и директивы (включая нестандартные директивы `paIf` и `paFor`); результат показан на рис. 17.10. Ключевое различие проявляется не во внешнем виде таблицы, а в том, что теперь таблицей управляет специализированный компонент.

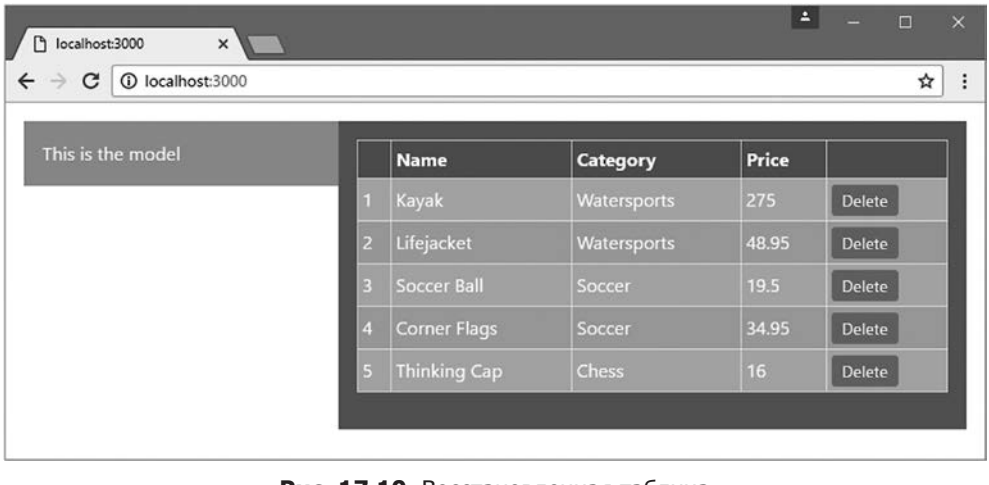


Рис. 17.10. Восстановленная таблица

Использование выходных свойств для координации между компонентами

Дочерние компоненты могут использовать выходные свойства, которые определяют нестандартные события, сигнализирующие о важных изменениях и позволяющие родительскому компоненту реагировать на их возникновение. В листинге 17.14 показано добавление в компонент формы выходного свойства, которое инициируется при создании нового объекта `Product`.

Листинг 17.14. Определение выходного свойства в файле productForm.component.ts

```
import { Component, Output, EventEmitter } from "@angular/core";
import { Product } from "./product.model";
import { ProductFormGroup } from "./form.model";

@Component({
    selector: "paProductForm",
    templateUrl: "app/productForm.component.html"
})
export class ProductFormComponent {
    form: ProductFormGroup = new ProductFormGroup();
    newProduct: Product = new Product();
    formSubmitted: boolean = false;

    @Output("paNewProduct")
    newProductEvent = new EventEmitter<Product>();

    submitForm(form: any) {
        this.formSubmitted = true;
        if (form.valid) {
            this.newProductEvent.emit(this.newProduct);
            this.newProduct = new Product();
            this.form.reset();
            this.formSubmitted = false;
        }
    }
}
```

Выходное свойство называется `newProductEvent`; событие инициируется компонентом при вызове метода `submitForm`. Если не считать выходного свойства, изменения в листинге базируются на логике корневого контроллера, который ранее управлял формой. Удалите встроенный шаблон, создайте файл с именем productForm.component.html в папке app и восстановите в нем форму HTML (листинг 17.15).

Листинг 17.15. Содержимое файла productForm.component.html в папке app

```
<form novalidate [formGroup]="form" (ngSubmit)="submitForm(form)">
    <div class="form-group" *ngFor="let control of form.productControls">
        <label>{{control.label}}</label>
        <input class="form-control"
            [(ngModel)]="newProduct[control.modelProperty]"
            name="{{control.modelProperty}}"
            formControlName="{{control.modelProperty}}" />
        <ul class="text-danger list-unstyled"
                *ngIf="(formSubmitted || control.dirty) && !control.valid">
            <li *ngFor="let error of control.getValidationMessages()">
                {{error}}
            </li>
        </ul>
    </div>
    <button class="btn btn-primary" type="submit"
        [disabled]="formSubmitted && !form.valid"
```

```
        [class.btn-secondary]="formSubmitted && !form.valid">
            Create
    </button>
</form>
```

В разметку, используемую для определения формы, никакие изменения вносить не нужно. Как и в случае со входным свойством, управляющий элемент дочернего компонента служит мостом к родительскому компоненту, который может зарегистрировать свой интерес к получению нестандартного события (листинг 17.16).

Листинг 17.16. Регистрация нестандартного события в файле template.html

```
<div class="col-xs-4 p-a-1">
    <paProductForm (paNewProduct)="addProduct($event)"></paProductForm>
</div>
<div class="col-xs-8 p-a-1">
    <paProductTable [model]="model"></paProductTable>
</div>
```

Новая привязка обрабатывает нестандартное событие, передавая объект события методу `addProduct`. Дочерний компонент отвечает за управление элементами формы и проверку их содержимого. Если данные проходят проверку, инициируется нестандартное событие и выражение привязки данных вычисляется в контексте родительского компонента, метод `addProduct` которого добавляет новый объект в модель. Так как доступ к модели предоставляется дочернему компоненту таблицы через его входное свойство, новые данные выводятся для пользователя (рис. 17.11).

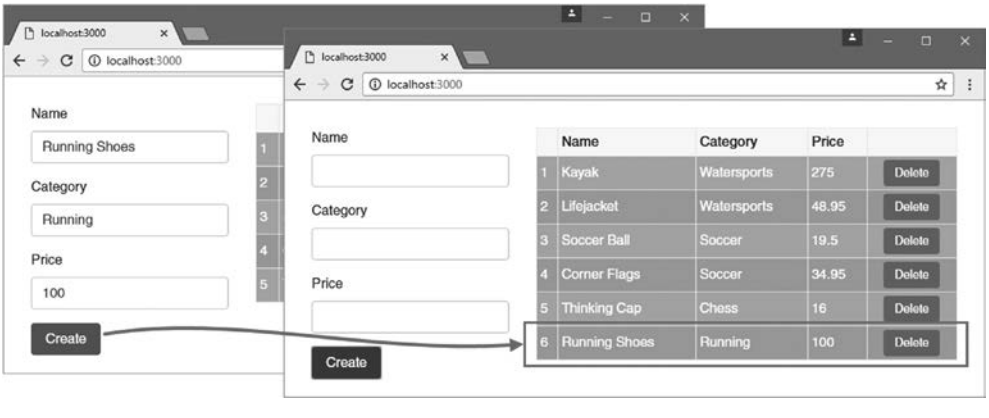


Рис. 17.11. Использование нестандартного события в дочернем компоненте

Проецирование контента управляющего элемента

Если управляющий элемент компонента содержит контент, его можно включить в шаблон специальным элементом `ng-content`. Этот механизм, называемый *проецированием контента*, позволяет создавать компоненты, объединяющие контент шаблона с контентом управляющего элемента. Добавьте файл toggleView.

component.ts в папку app и включите в него определение компонента из листинга 17.17.

Листинг 17.17. Содержимое файла toggleView.component.ts в папке app

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

@Component({
    selector: "paToggleView",
    templateUrl: "app/toggleView.component.html"
})
export class PaToggleView {

    showContent: boolean = true;
}
```

Компонент определяет свойство `showContent`, которое будет использоваться для определения того, должен ли контент управляющего элемента отображаться в шаблоне. Чтобы предоставить шаблон, создайте файл toggleView.component.html в папке app и добавьте элементы из листинга 17.18.

Листинг 17.18. Содержимое файла toggleView.component.html в папке app

```
<div class="checkbox">
    <label>
        <input type="checkbox" [(ngModel)]="showContent" />
        Show Content
    </label>
</div>
<ng-content *ngIf="showContent"></ng-content>
```

В этой разметке важную роль играет элемент `ng-content`, который Angular заменит контентом управляющего элемента.

К элементу `ng-content` применяется директива `ngIf`, чтобы он отображался только в том случае, если установлен флажок в шаблоне. В листинге 17.19 компонент регистрируется в модуле Angular.

Листинг 17.19. Регистрация компонента в файле app.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { ProductComponent } from "./component";
import { FormsModule, ReactiveFormsModule  } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
import { PaModel } from "./twoway.directive";
import { PaStructureDirective } from "./structure.directive";
import { PaIteratorDirective } from "./iterator.directive";
import { PaCellColor } from "./cellColor.directive";
import { PaCellColorSwitcher } from "./cellColorSwitcher.directive";
import { ProductTableComponent } from "./productTable.component";
import { ProductFormComponent } from "./productForm.component";
import { PaToggleView } from "./toggleView.component";
```

```
@NgModule({
    imports: [BrowserModule, FormsModule, ReactiveFormsModule],
    declarations: [ProductComponent, PaAttrDirective, PaModel,
        PaStructureDirective, PaIteratorDirective,
        PaCellColor, PaCellColorSwitcher, ProductTableComponent,
        ProductFormComponent, PaToggleView],
    bootstrap: [ProductComponent]
})
export class AppModule { }
```

Остается применить новый компонент к управляющему элементу, содержащему контент (листинг 17.20).

Листинг 17.20. Добавление управляющего элемента с контентом в файл template.html

```
<div class="col-xs-4 p-a-1">
    <paProductForm (paNewProduct)="addProduct($event)"></paProductForm>
</div>
<div class="col-xs-8 p-a-1">
    <paToggleView>
        <paProductTable [model]="model"></paProductTable>
    </paToggleView>
</div>
```

Элемент `paToggleView` является управляющим для нового компонента, и он содержит элемент `paProductTable`, который применяет компонент, создающий таблицу товаров. В результате появляется флажок, управляющий видимостью таблицы (рис. 17.12). Новый компонент не располагает информацией о контенте управляющего элемента, а его включение в шаблон возможно только через элемент `ng-content`.

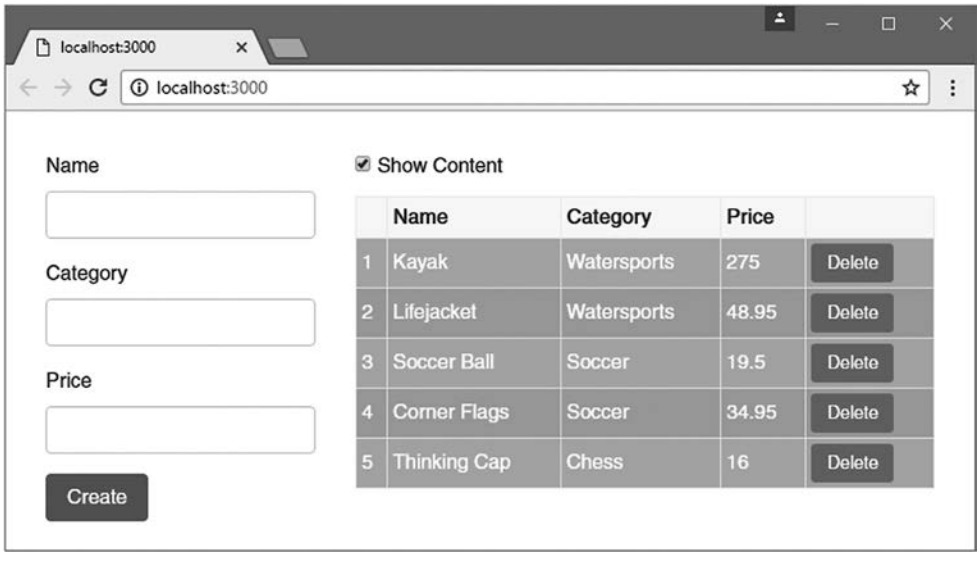


Рис. 17.12. Включение контента управляющего элемента в шаблон

Завершение реструктуризации компонента

Функциональность, которая ранее содержалась в корневом компоненте, распределяется между новыми дочерними компонентами. Остается только «почистить» корневой компонент и удалить код, который стал лишним (листинг 17.21).

Листинг 17.21. Удаление лишнего кода из файла component.ts

```
import { ApplicationRef, Component } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";
import { ProductFormGroup } from "./form.model";

@Component({
    selector: "app",
    templateUrl: "app/template.html"
})
export class ProductComponent {
    model: Model = new Model();

    addProduct(p: Product) {
        this.model.saveProduct(p);
    }
}
```

Многие обязанности корневого компонента были переданы в другие части приложения. Из исходного списка, приведенного в начале главы, остались всего два пункта; компонент:

- предоставляет Angular точку входа приложения (как корневой компонент);
- предоставляет доступ к модели данных приложения для использования в привязках данных.

Остальные обязанности взяли на себя дочерние компоненты. Они предоставляют автономные блоки функциональности — более простые, удобные в разработке и сопровождении и пригодные для повторного использования в случае необходимости.

Использование стилей компонентов

Компоненты могут определять стили, которые применяются только к контенту в их шаблонах. Это позволяет применять стилевое оформление контента на уровне компонента без влияния стилей, определяемых родителями или предками более высокого уровня, и без влияния на контент дочерних элементов и других потомков. Стили могут определяться как встроенные при помощи свойства `styles` декоратора `@Component` (листинг 17.22).

Листинг 17.22. Определение встроенных стилей в файле productForm.component.ts

```
import { Component, Output, EventEmitter } from "@angular/core";
import { Product } from "./product.model";
import { ProductFormGroup } from "./form.model";

@Component({
```

```
    selector: "paProductForm",
    templateUrl: "app/productForm.component.html",
    styles: ["div { background-color: lightgreen"]
})
export class ProductFormComponent {
    form: ProductFormGroup = new ProductFormGroup();
    newProduct: Product = new Product();
    formSubmitted: boolean = false;

   @Output("paNewProduct")
    newProductEvent = new EventEmitter<Product>();

    submitForm(form: any) {
        this.formSubmitted = true;
        if (form.valid) {
            this.newProductEvent.emit(this.newProduct);
            this.newProduct = new Product();
            this.form.reset();
            this.formSubmitted = false;
        }
    }
}
```

Свойству `styles` присваивается массив, каждый элемент которого содержит селектор CSS и одно или несколько свойств. В листинге я задал стили, назначающие элементам `div` светло-зеленый цвет фона. И хотя элементы `div` содержатся в объединенном документе HTML, этот стиль будет влиять только на элементы в шаблоне компонента, определяющего их, — в нашем примере это компонент формы (рис. 17.13).

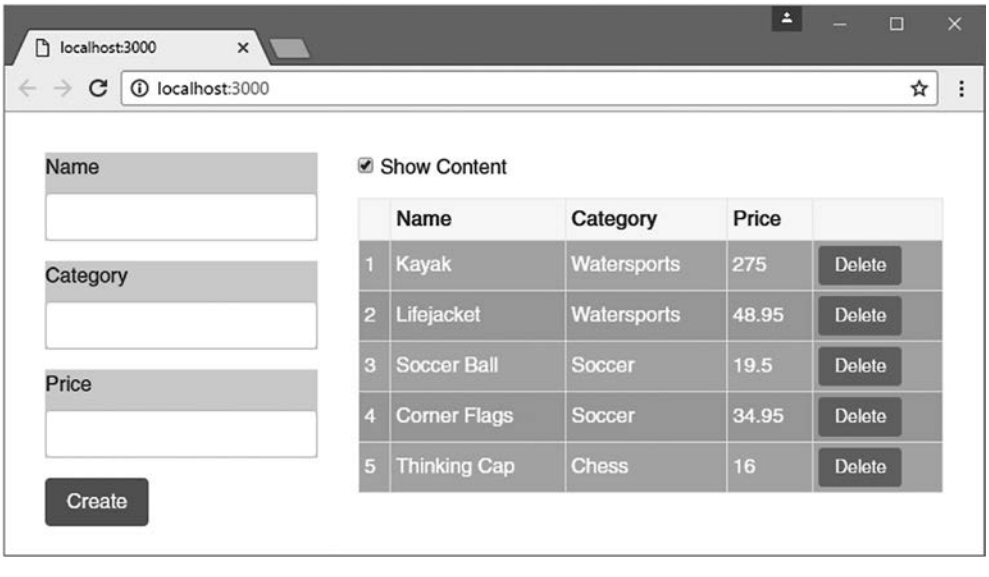


Рис. 17.13. Определение встроенных стилей компонентов

ПРИМЕЧАНИЕ

Стили, определяемые в секции head документа HTML, относятся ко всем элементам даже в том случае, если компонент определяет свои собственные стили. Именно по этой причине контент по-прежнему оформляется с использованием Bootstrap.

Определение внешних стилей компонентов

Встроенные стили обладают теми же достоинствами и недостатками, что и встроенные шаблоны: они просты, а вся информация хранится в одном файле. При этом они иногда плохо читаются, создают сложности с управлением и приводят в замешательство редакторы кода.

Альтернативное решение — определение стилей в отдельном файле и связывание их с компонентом при помощи свойства `styleUrls` в его декораторе. Имена внешних стилевых файлов строятся по той же схеме, что и имена шаблонов и файлов кода. Создайте файл productForm.component.css в папке app и включите в него определения стилей из листинга 17.23.

Листинг 17.23. Содержимое файла productForm.component.css в папке app

```
div {
    background-color: lightcoral;
}
```

Это тот же стиль, который был определен как встроенный, но с другим цветовым значением — оно подтверждает, что этот стиль CSS используется компонентом. В листинге 17.24 декоратор компонента обновляется с указанием стилевого файла.

Листинг 17.24. Использование внешнего стиля компонента в файле productForm.component.ts

```
import { Component, Output, EventEmitter } from "@angular/core";
import { Product } from "./product.model";
import { ProductFormGroup } from "./form.model";

@Component({
    selector: "paProductForm",
    templateUrl: "app/productForm.component.html",
    styleUrls: ["app/productForm.component.css"]
})
export class ProductFormComponent {

    // ...Свойства и методы класса опущены для краткости...

}
```

Свойству `styleUrls` задается массив строк, каждая из которых определяет файл CSS. На рис. 17.14 показан эффект добавления внешнего стилевого файла.

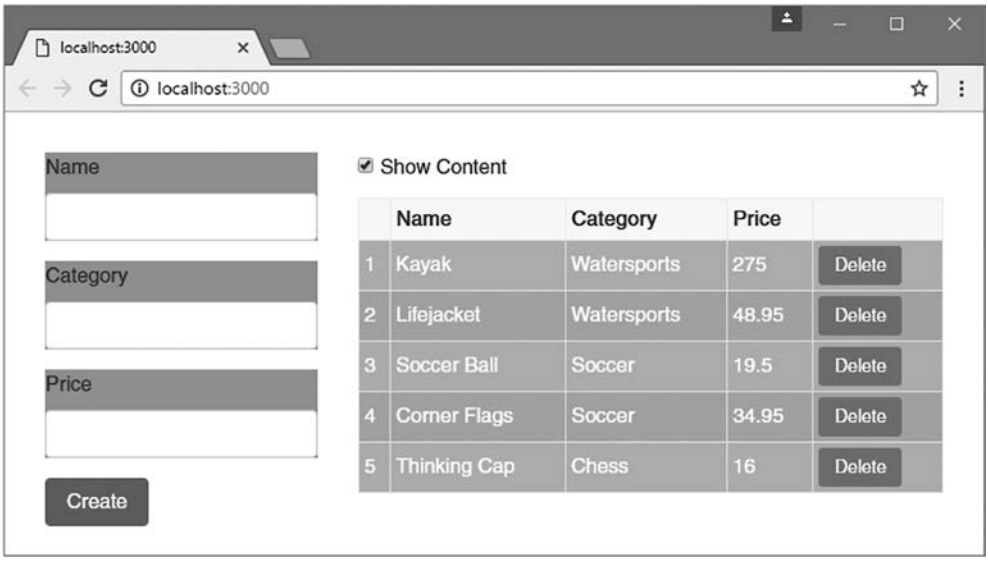


Рис. 17.14. Определение внешнего стиля компонента

Расширенные возможности стилей

Определение стилей в компонентах — полезная возможность, однако вы не всегда будете получать именно те результаты, на которые рассчитываете. Некоторые расширенные возможности помогут вам в управлении стилями компонентов.

Инкапсуляция представления

По умолчанию стили, относящиеся к конкретным компонентам, реализуются написанием разметки CSS, которая применяется к компоненту. Angular добавляет эту разметку во все элементы верхнего уровня, содержащиеся в шаблоне компонента. Если вы проанализируете DOM с использованием средств разработчика F12, то увидите, что содержимое внешнего файла CSS из листинга 17.23 было добавлено в элемент `head` документа HTML в слегка измененном виде:

```
...
<style>
div[_ngcontent-qsg-2] {
    background-color: lightcoral;
}
</style>
...
```

Селектор был изменен так, чтобы он находил элементы `div` с атрибутом `_ngcontent-qsg-2` (в вашем браузере имя может выглядеть иначе, потому что Angular генерирует имя атрибута динамически).

Чтобы разметка CSS в элементе `style` влияла только на элементы HTML, находящиеся под управлением компонента, элементы шаблона изменяются таким образом, чтобы они содержали тот же автоматически сгенерированный атрибут:

```
...
<div _ngcontent-qsg-2="" class="form-group">
        <label _ngcontent-qsg-2="">Name</label>
        <input _ngcontent-qsg-2="" class="form-control ng-untouched ng-pristine
                ng-invalid" ng-reflect-name="name" name="name">
</div>
...
```

Это поведение, называемое *инкапсуляцией представления* компонента, применяется Angular для эмуляции функциональности *теневой модели DOM*, которая позволяет изолировать секции DOM в границах их собственной области видимости; это означает, что JavaScript, стили и шаблоны могут применяться к отдельным частям документа HTML. Причина, по которой Angular эмулирует это поведение, заключается в том, что оно реализовано лишь в малой части браузеров (на момент написания книги теневая модель DOM поддерживалась только Google Chrome и новейшими версиями Safari и Opera). Существуют два других режима инкапсуляции, которые задаются при помощи свойства `encapsulation` декоратора `@Component`.

ПРИМЕЧАНИЕ

Дополнительную информацию о теневой модели DOM можно найти по адресу http://developer.mozilla.org/en-US/docs/Web/Web_Components/Shadow_DOM. Список браузеров, поддерживающих теневую модель DOM, доступен по адресу http://caniuse.com/#feat=shadowdom.

Свойству `encapsulation` задаются значения из перечисления `ViewEncapsulation`, определяемого в модуле `@angular/core`. Значения, определяемые в этом перечислении, описаны в табл. 17.4.

Таблица 17.4. Значения ViewEncapsulation

Имя	Описание
Emulated	С этим значением Angular эмулирует теневую модель DOM посредством записи контента и стилей для добавления атрибутов (см. выше). Это поведение используется по умолчанию, если значение encapsulation не задано в декораторе @Component
Native	Если задано это значение, Angular использует поддержку теневой модели DOM в браузере. Оно работает в том случае, если браузер реализует теневую модель DOM, или же при использовании полизаполнений
None	С этим значением Angular просто добавляет неизменные стили CSS в секцию head документа HTML; предполагается, что браузер сам разберется, как применять стили по нормальным правилам приоритета CSS

Значения `Native` и `None` следует использовать с осторожностью. Поддержка теневой модели DOM настолько ограниченна, что режим `Native` есть смысл использовать только в том случае, если вы используете библиотеку полизаполнения для обеспечения совместимости в других браузерах. К сожалению, существующие библиотеки полизаполнений (например, доступная по адресу *http://github.com/webcomponents/webcomponentsjs*) обеспечивают нестабильные результаты и не работают со всеми браузерами, поддерживаемыми Angular.

В режиме `None` все стили, определяемые компонентами, добавляются в секцию `head` документа HTML, а браузер сам должен разобраться, как их применить. С одной стороны, это решение подходит для любого браузера, с другой — результаты непредсказуемы, а стили, определяемые разными компонентами, не изолируются друг от друга.

Для полноты картины в листинге 17.25 свойству `encapsulation` задается значение по умолчанию `Emulated`, которое работает во всех браузерах, поддерживаемых Angular, без необходимости использования полизаполнений.

Листинг 17.25. Назначение инкапсуляции представления в файле productForm.component.ts

```
import { Component, Output, EventEmitter, ViewEncapsulation } from "@angular/core";
import { Product } from "./product.model";
import { ProductFormGroup } from "./form.model";

@Component({
    selector: "paProductForm",
    templateUrl: "app/productForm.component.html",
    styleUrls: ["app/productForm.component.css"],
    encapsulation: ViewEncapsulation.Emulated
})
export class ProductFormComponent {
    form: ProductFormGroup = new ProductFormGroup();
    newProduct: Product = new Product();
    formSubmitted: boolean = false;

    @Output("paNewProduct")
    newProductEvent = new EventEmitter<Product>();

    submitForm(form: any) {
        this.formSubmitted = true;
        if (form.valid) {
            this.newProductEvent.emit(this.newProduct);
            this.newProduct = new Product();
            this.form.reset();
            this.formSubmitted = false;
        }
    }
}
```

Использование селекторов CSS теневой модели DOM

Использование теневой модели DOM означает, что существуют границы, за пределами которых обычные селекторы CSS не работают. Для решения этой проблемы существуют специальные селекторы CSS, используемые при работе со стилями, зависящими от теневой модели DOM (даже при эмуляции). Эти селекторы описаны в табл. 17.5, а их использование продемонстрировано ниже.

Таблица 17.5. Селекторы CSS теневой модели DOM

Имя	Описание
:host	Селектор используется для идентификации управляющего элемента компонента
:host-context(classSelector)	Селектор используется для идентификации предков управляющего элемента, которые принадлежат к заданному классу
/deep/ или >>>	Селектор используется родительским компонентом для определения стилей, влияющих на элементы шаблонов дочернего компонента. Селектор должен использоваться только в том случае, если свойству encapsulation декоратора @Component задано значение emulated (см. раздел «Инкапсуляция представления»)

Выбор управляющего элемента

Управляющий элемент компонента находится за пределами шаблона; это означает, что селекторы в его стилях применяются только к элементам, содержащимся в управляющем элементе, но не к самому элементу. Проблема может быть решена при помощи селектора `:host`, который обозначает управляющий элемент. В листинге 17.26 определяется стиль, который применяется только в том случае, когда указатель мыши наводится на управляющий элемент; он задается объединением селекторов `:host` и `:hover`.

Листинг 17.26. Идентификация управляющего элемента компонента в файле productForm.component.css

```
div {
    background-color: lightcoral;
}
:host:hover {
    font-size: 25px;
}
```

Когда указатель мыши находится над управляющим элементом, его свойство `font-size` задается равным `25px`; размер текста всех элементов формы увеличивается до 25 пунктов (рис. 17.15).

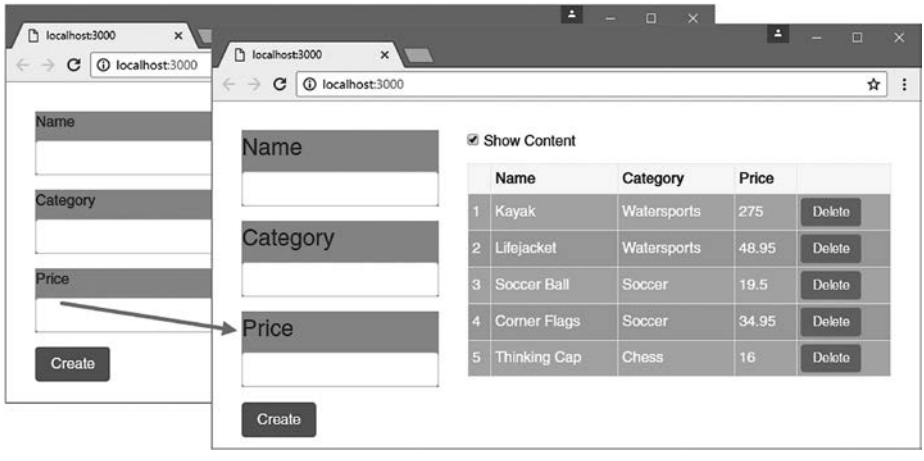


Рис. 17.15. Выбор управляющего элемента в стиле компонента

Выбор предков управляющего элемента

Селектор `:host-context` используется для стилевого оформления элементов в шаблоне компонента на основании принадлежности предков управляющего элемента (находящихся за пределами шаблона) к классу. Этот селектор более ограничен, чем `:host`; он не может использоваться для определения чего-либо, кроме селектора класса, и не поддерживает поиск по типу тега, атрибуту и т. д. В листинге 17.27 продемонстрировано использование селектора `:host-context`.

Листинг 17.27. Выбор предков управляющего элемента в файле productForm.component.css

```
div {
    background-color: lightcoral;
}
:host:hover {
    font-size: 25px;
}
:host-context(.angularApp) input {
    background-color: lightgray;
}
```

Селектор в листинге задает свойству `background-color` элементов `input` в шаблоне компонента значение `lightgrey` только в том случае, если один из элементов-предков управляющего элемента принадлежит к классу с именем `angularApp`. В листинге 17.28 я добавил в файле index.html элемент app (управляющий элемент для корневого компонента) в класс `angularApp`.

Листинг 17.28. Добавление управляющего элемента корневого компонента в файл index.html

```
<!DOCTYPE html>
<html>
<head>
```

```
    <title></title>
    <meta charset="utf-8" />
    <script src="node_modules/classlist.js/classList.min.js"></script>
    <script src="node_modules/core-js/client/shim.min.js"></script>
    <script src="node_modules/zone.js/dist/zone.min.js"></script>
    <script src="node_modules/reflect-metadata/Reflect.js"></script>
    <script src="node_modules/systemjs/dist/system.src.js"></script>
    <script src="systemjs.config.js"></script>
    <script>
      System.import("app/main").catch(function(err){ console.error(err); });
    </script>
    <link href="node_modules/bootstrap/dist/css/bootstrap.min.css"
        rel="stylesheet" />
</head>
<body class="m-a-1">
    <app class="angularApp"></app>
</body>
</html>
```

На рис. 17.16 показан эффект применения селектора до и после изменений в листинге 17.28.

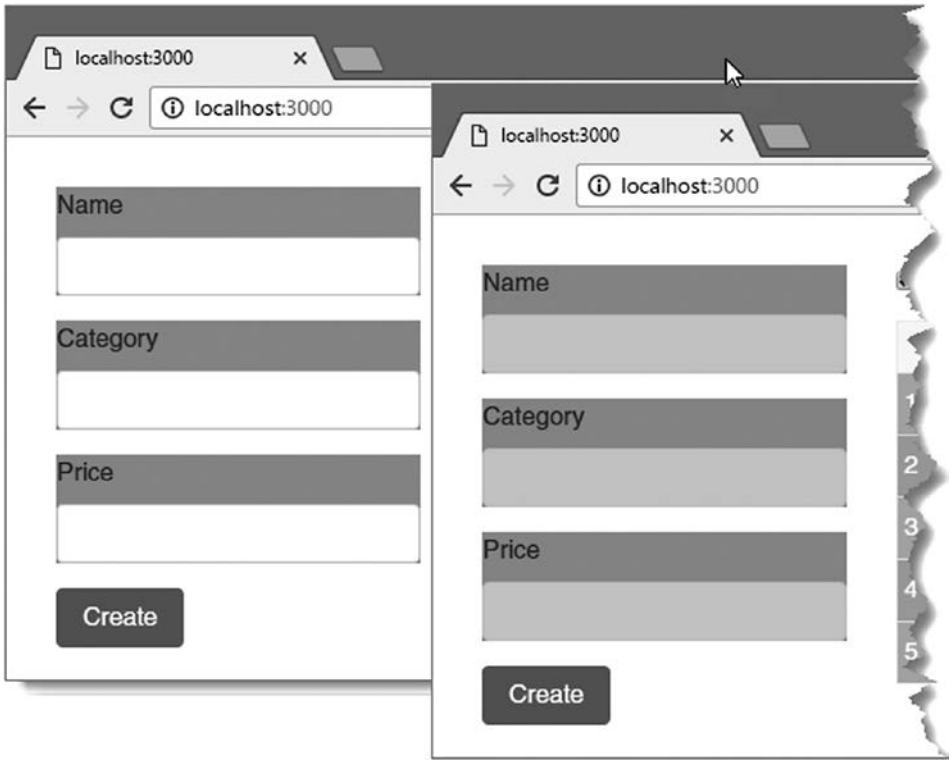


Рис. 17.16. Выбор предков управляющего элемента

Продвижение стиля в шаблон дочернего компонента

Стили, определенные компонентом, не применяются автоматически к элементам шаблонов дочерних компонентов. Для демонстрации в листинге 17.29 стиль добавляется в декоратор `@Component` корневого компонента.

Листинг 17.29. Определение стилей в файле component.ts

```
import { ApplicationRef, Component } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";
import { ProductFormGroup } from "./form.model";

@Component({
    selector: "app",
    templateUrl: "app/template.html",
    styles: ["div { border: 2px black solid;  font-style:italic }"]
})
export class ProductComponent {
    model: Model = new Model();

    addProduct(p: Product) {
        this.model.saveProduct(p);
    }
}
```

Селектор находит все элементы `div`, применяет рамку и изменяет стиль фона. Результат показан на рис. 17.17.

Некоторые свойства стилей CSS, такие как `font-style`, наследуются по умолчанию; это означает, что задание такого свойства в родительском компоненте

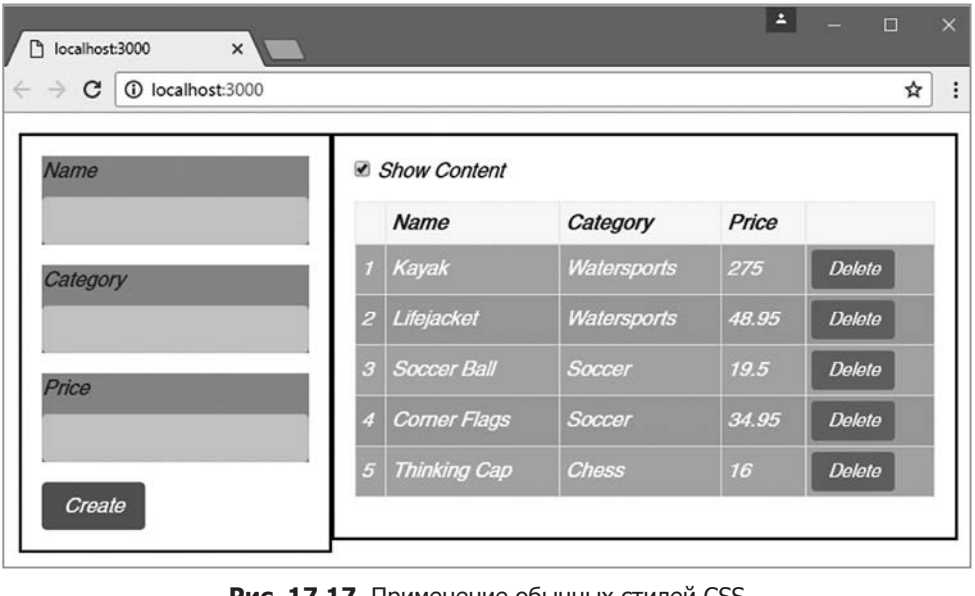


Рис. 17.17. Применение обычных стилей CSS

повлияет на элементы шаблонов дочерних компонентов, потому что браузер автоматически применяет стиль.

Другие свойства, такие как `border`, не наследуются по умолчанию, и задание такого свойства в родительском компоненте не распространяется на шаблоны дочерних компонентов, если только не используются селекторы `/deep/` или `>>>` (листинг 17.30). (Эти селекторы являются синонимами и работают одинаково.)

Листинг 17.30. Продвижение стиля в шаблоны дочерних компонентов в файле component.ts

```
import { ApplicationRef, Component } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";
import { ProductFormGroup } from "./form.model";

@Component({
    selector: "app",
    templateUrl: "app/template.html",
    styles: ["/deep/ div { border: 2px black solid;  font-style:italic }"]
})
export class ProductComponent {
    model: Model = new Model();

    addProduct(p: Product) {
        this.model.saveProduct(p);
    }
}
```

Селектор стиля использует конструкцию `/deep/` для продвижения стилей в шаблоны дочерних компонентов; это означает, что всем элементам `div` будет назначена рамка (рис. 17.18).

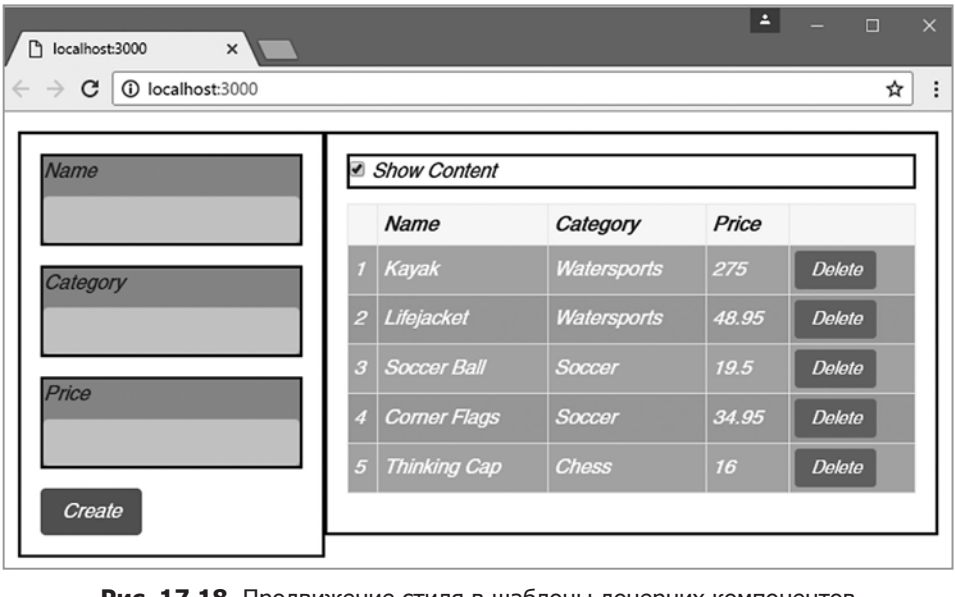


Рис. 17.18. Продвижение стиля в шаблоны дочерних компонентов

Запрос информации о контенте шаблона

Компоненты могут запрашивать контент шаблонов для получения экземпляров директив или компонентов, называемых *дочерними элементами представления* (view children). Такие запросы напоминают запросы контента дочерних директив, описанные в главе 16, но между ними есть ряд важных различий.

В листинге 17.21 в компонент, управляющий таблицей, добавляется код запроса директивы `PaCellColor`, созданной для демонстрации запросов контента директив. Директива по-прежнему регистрируется в модуле Angular и выбирает элементы `td`, поэтому Angular применяет ее к ячейкам в контенте табличного компонента.

Листинг 17.31. Выбор дочерних элементов представления в файле productTable.component.ts

```
import { Component, Input, ViewChildren, QueryList } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";
import { PaCellColor } from "./cellColor.directive";

@Component({
    selector: "paProductTable",
    templateUrl: "app/productTable.component.html"
})
export class ProductTableComponent {

    @Input("model")
    dataModel: Model;

    getProduct(key: number): Product {
        return this.dataModel.getProduct(key);
    }

    getProducts(): Product[] {
        return this.dataModel.getProducts();
    }

    deleteProduct(key: number) {
        this.dataModel.deleteProduct(key);
    }

    showTable: boolean = true;

    @ViewChildren(PaCellColor)
    viewChildren: QueryList<PaCellColor>;

    ngAfterViewInit() {
        this.viewChildren.changes.subscribe(() => {
            this.updateViewChildren();
        });
        this.updateViewChildren();
    }
```

```
    private updateViewChildren() {
        setTimeout(() => {
            this.viewChildren.forEach((child, index) => {
                child.setColor(index % 2 ? true : false);
            })
        }, 0);
    }
}
```

Существуют два декоратора свойств, используемые для запроса директив или компонентов, определенных в шаблоне (табл. 17.6).

Таблица 17.6. Декораторы свойств запроса дочерних элементов представления

Имя	Описание
@ViewChild(class)	Декоратор приказывает Angular запросить объект первой директивы или компонента заданного типа и задать его свойству. Имя класса может быть заменено переменной шаблона. Множественные классы или шаблоны разделяются запятыми
@ViewChildren(class)	Декоратор задает все объекты директив и компонентов заданного типа свойству. Вместо классов могут использоваться переменные шаблона, множественные значения разделяются запятыми. Результаты передаются в объекте QueryList (см. главу 16)

В листинге декоратор `@ViewChildren` используется для выбора всех объектов `PaCellColor` из шаблона компонента. Помимо разных декораторов свойств, компоненты содержат два метода жизненного цикла, которые используются для передачи информации о том, как обрабатывается шаблон (табл. 17.7).

Таблица 17.7. Дополнительные методы жизненного цикла компонента

Имя	Описание
ngAfterViewInit	Метод вызывается при инициализации представления компонента. Результаты запросов представления задаются до вызова этого метода
ngAfterViewChecked	Метод вызывается после проверки представления компонента как части процесса обнаружения изменений

В листинге реализуется метод `ngAfterViewInit`, который проверяет, что среда Angular обработала шаблон компонента и задала результат запроса. Внутри метода выполняется исходный вызов метода `updateViewChildren`, который работает с объектами `PaCellColor`, и задается функция, которая будет вызываться при изменении результатов запроса, для чего используется свойство `QueryList.changes` (см. главу 16). Дочерние элементы представления обновляются внутри вызова функции `setTimeout`, как объяснялось в главе 16. В результате изменяется цвет каждой второй таблицы ячейки (рис. 17.19).

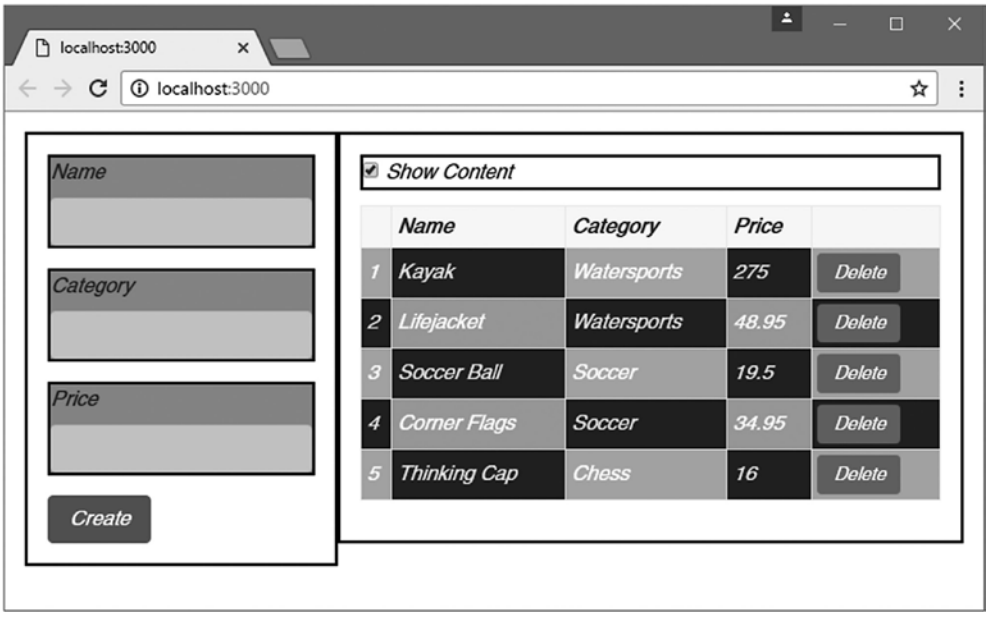


Рис. 17.19. Запрос дочерних элементов представления

ПРИМЕЧАНИЕ

Возможно, вам потребуется объединить дочерние элементы представления и запросы дочернего контента, если вы использовали элемент ng-content. Контент, определяемый в шаблоне, запрашивается способом, представленным в листинге 17.31, но проецированный контент — заменяющий элемент ng-content — запрашивается с использованием запросов дочерних элементов, описанных в главе 16.

Итоги

В этой главе мы вернулись к теме компонентов и рассмотрели, как объединить всю функциональность директив со способностью предоставлять собственные шаблоны. Я объяснил, как структурировать приложение для создания небольших модульных компонентов и как компоненты могут координировать свою работу при помощи входных и выходных свойств. Я также показал, как компоненты могут определять стили CSS, которые применяются только к их собственным шаблонам, но не действуют на другие части приложения. В следующей главе я расскажу о каналах, используемых для подготовки данных, предназначенных для отображения в шаблонах.

18 Использование и создание каналов

Каналы (pipes) — небольшие фрагменты кода, которые преобразуют значения данных для отображения в шаблонах. Каналы позволяют определить логику преобразования в автономных классах, чтобы их можно было последовательно применять в приложении. В табл. 18.1 каналы представлены в контексте.

Таблица 18.1. Каналы в контексте

Вопрос	Ответ
Что это такое?	Каналы — классы, предназначенные для подготовки данных, отображаемых для пользователя
Для чего они нужны?	Каналы позволяют определить логику подготовки данных в одном классе, который может использоваться в разных местах приложения; тем самым обеспечивается логическая целостность представления данных
Как они используются?	Декоратор @Pipe применяется к классу и используется для задания имени, по которому канал может использоваться в шаблоне
Есть ли у них недостатки или скрытые проблемы?	Каналы должны быть простыми и ориентированными на подготовку данных. Иногда возникает соблазн включить в них функциональность, за которую должны отвечать другие структурные блоки приложения, например директивы или компоненты
Есть ли альтернативы?	Код подготовки данных можно реализовать в компонентах или директивах, но это усложнит его повторное использование в других частях приложения

В табл. 18.2 приведена краткая сводка материала главы.

Таблица 18.2. Сводка материала главы

Проблема	Решение	Листинг
Форматирование значения данных для включения в шаблон	Используйте канал в выражении привязки данных	1–8
Создание нестандартного канала	Примените декоратор @Pipe к классу	9–11

Таблица 18.2 (*окончание*)

Проблема	Решение	Листинг
Форматирование значения данных с использованием нескольких каналов	Объедините имена каналов символом |	12
Определение необходимости повторного вычисления вывода канала	Используйте свойство pure декоратора @Pipe	13–16
Форматирование числовых значений	Используйте канал number	17, 18
Форматирование денежных сумм	Используйте канал currency	19, 20
Форматирование процентов	Используйте канал percent	21–24
Изменение регистра строк	Используйте каналы uppercase и lowercase	25
Сериализация объектов в формате JSON	Используйте канал json	26
Выбор элементов из массива	Используйте канал slice	27

Подготовка проекта

В этой главе мы продолжим использовать проект example, который был создан в главе 11 и дорабатывался и расширялся в последующих главах. В последних примерах предыдущей главы стили компонентов и запросы дочерних элементов представления оставили приложение с чересчур пестрым оформлением, которое нужно привести в норму. В листинге 18.1 отключаются встроенные стили компонентов, примененные к элементам формы.

Листинг 18.1. Отключение стилей CSS в файле productForm.component.ts

```
import { Component, Output, EventEmitter, ViewEncapsulation } from "@angular/core";
import { Product } from "./product.model";
import { ProductFormGroup } from "./form.model";

@Component({
    selector: "paProductForm",
    templateUrl: "app/productForm.component.html",
    //styleUrls: ["app/productForm.component.css"],
    //encapsulation: ViewEncapsulation.Emulated
})
export class ProductFormComponent {
    form: ProductFormGroup = new ProductFormGroup();
    newProduct: Product = new Product();
    formSubmitted: boolean = false;

    @Output("paNewProduct")
    newProductEvent = new EventEmitter<Product>();

    submitForm(form: any) {
        this.formSubmitted = true;
```

```
        if (form.valid) {
            this.newProductEvent.emit(this.newProduct);
            this.newProduct = new Product();
            this.form.reset();
            this.formSubmitted = false;
        }
    }
}
```

Чтобы отключить шахматную окраску ячеек таблицы, я изменил селектор директивы `PaCellColor`, чтобы он соответствовал атрибуту, который в настоящее время не применяется к элементам HTML (листинг 18.2).

Листинг 18.2. Изменение селектора в файле cellColor.directive.ts

```
import { Directive, HostBinding } from "@angular/core";

@Directive({
    selector: "td[paApplyColor]"
})
export class PaCellColor {

    @HostBinding("class")
    bgClass: string = "";

    setColor(dark: Boolean) {
        this.bgClass = dark ? "bg-inverse" : "";
    }
}
```

В листинге 18.3 отключаются `deep`-стили, определяемые корневым компонентом.

Листинг 18.3. Отключение стилей CSS в файле component.ts

```
import { ApplicationRef, Component } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";
import { ProductFormGroup } from "./form.model";

@Component({
    selector: "app",
    templateUrl: "app/template.html",
    //styles: ["/deep/ div { border: 2px black solid;  font-style:italic }"]
})
export class ProductComponent {
    model: Model = new Model();

    addProduct(p: Product) {
        this.model.saveProduct(p);
    }
}
```

Следующее изменение в коде примера — упрощение класса `ProductTableComponent` с удалением свойств и методов, которые стали лишними (листинг 18.4).

Листинг 18.4. Упрощение кода в файле productTable.component.ts

```
import { Component, Input, ViewChildren, QueryList } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";

@Component({
    selector: "paProductTable",
    templateUrl: "app/productTable.component.html"
})
export class ProductTableComponent {

    @Input("model")
    dataModel: Model;

    getProduct(key: number): Product {
        return this.dataModel.getProduct(key);
    }

    getProducts(): Product[] {
        return this.dataModel.getProducts();
    }

    deleteProduct(key: number) {
        this.dataModel.deleteProduct(key);
    }
}
```

Наконец, из шаблона корневого компонента удаляется один из элементов компонента для блокировки флажка, управляющего отображением и сокрытием таблицы (листинг 18.5).

Листинг 18.5. Упрощение элементов в файле template.html

```
<div class="col-xs-4 p-a-1">
    <paProductForm (paNewProduct)="addProduct($event)"></paProductForm>
</div>
<div class="col-xs-8 p-a-1">
    <paProductTable [model]="model"></paProductTable>
</div>
```

Установка полизаполнения интернационализации

Некоторые встроенные каналы, предоставляемые Angular, зависят от API интернационализации, предоставляющего операции сравнения строк, а также форматирования чисел, дат и времени с учетом локального контекста. API поддерживается многими современными браузерами, но для некоторых браузеров, поддерживаемых Angular (включая старые версии Internet Explorer), требуется использование библиотеки полизаполнения. На момент написания книги самым полным полизаполнением для API интернационализации была библиотека Intl.js, которая в листинге 18.6 добавляется в набор зависимостей пакета.

Листинг 18.6. Добавление пакета в файл package.json

```
{
  "dependencies": {
    "@angular/common": "2.2.0",
    "@angular/compiler": "2.2.0",
    "@angular/core": "2.2.0",
    "@angular/platform-browser": "2.2.0",
    "@angular/platform-browser-dynamic": "2.2.0",
    "@angular/upgrade": "2.2.0",
    "@angular/forms": "2.2.0",
    "reflect-metadata": "0.1.8",
    "rxjs": "5.0.0-beta.12",
    "zone.js": "0.6.26",
    "core-js": "2.4.1",
    "classlist.js": "1.1.20150312",
    "systemjs": "0.19.40",
    "bootstrap": "4.0.0-alpha.4",
    "intl": "1.2.4"
  },

  "devDependencies": {
      "lite-server": "2.2.2",
      "typescript": "2.0.2",
      "typings": "1.3.2",
      "concurrently": "2.2.0"
  },

  "scripts": {
    "start": "concurrently \"npm run tscwatch\" \"npm run lite\" ",
    "tsc": "tsc",
    "tscwatch": "tsc -w",
    "lite": "lite-server",
    "typings": "typings",
    "postinstall": "typings install"
  }
}
```

Выполните следующую команду из папки example, чтобы загрузить и установить новый пакет:

```
npm install
```

В листинге 18.7 в основной документ HTML добавляется элемент `script` для загрузки библиотеки полизаполнения.

Листинг 18.7. Добавление элемента script в файл index.html

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
    <meta charset="utf-8" />
    <script src="node_modules/classlist.js/classList.min.js"></script>
```

```
    <script src="node_modules/core-js/client/shim.min.js"></script>
    <script src="node_modules/intl/dist/Intl.complete.js"></script>
    <script src="node_modules/zone.js/dist/zone.min.js"></script>
    <script src="node_modules/reflect-metadata/Reflect.js"></script>
    <script src="node_modules/systemjs/dist/system.src.js"></script>
    <script src="systemjs.config.js"></script>
    <script>
      System.import("app/main").catch(function(err){ console.error(err); });
    </script>
    <link href="node_modules/bootstrap/dist/css/bootstrap.min.css"
        rel="stylesheet" />
</head>
<body class="m-a-1">
    <app class="angularApp"></app>
</body>
</html>
```

Файл Intl.complete.js, добавленный элементом `script` в листинге 18.7, включает всю информацию локального контекста, доступную в библиотеке Intl.js; это удобно, но браузеру приходится загружать относительно большой файл. Инструкции по поводу избирательной загрузки данных локального контекста представлены на домашней странице библиотеки (*github.com/andyearnshaw/Intl.js*).

Выполните следующую команду из папки example, чтобы запустить компилятор TypeScript и сервер HTTP для разработки:

```
npm start
```

На экране появляется новое окно (или вкладка) браузера с контентом, показанным на рис. 18.1.

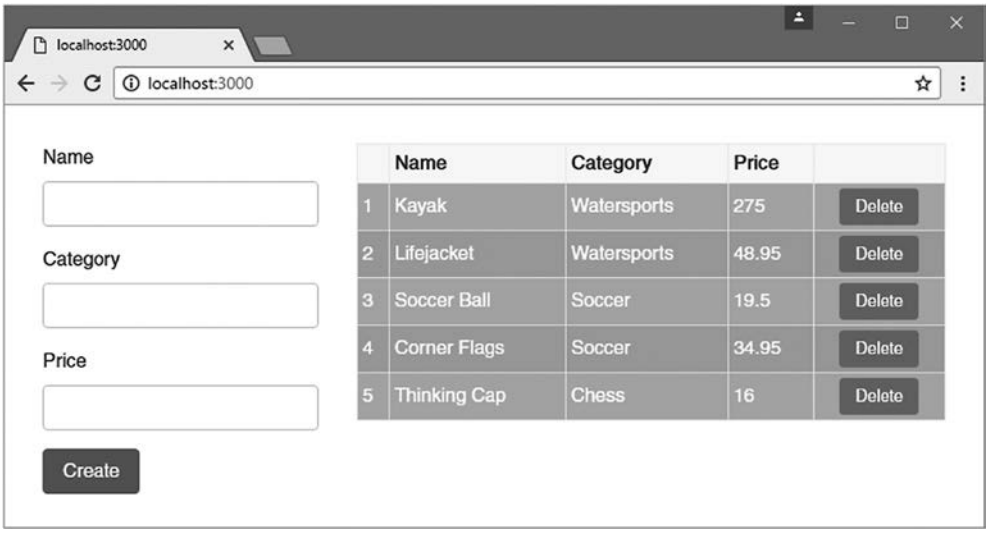


Рис. 18.1. Запуск приложения

Каналы

Каналы (pipes) — классы, которые преобразуют данные перед их получением директивой или компонентом. На первый взгляд задача не кажется особо важной, но каналы могут использоваться для простого и, что еще важнее, последовательного выполнения наиболее распространенных операций разработки.

В листинге 18.8 приведен простой пример: один из встроенных каналов используется для преобразования значений, выводимых в столбце Price таблицы приложения.

Листинг 18.8. Использование канала в файле productTable.component.html

```
<table class="table table-sm table-bordered table-striped">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th><th></th></tr>
    <tr *paFor="let item of getProducts(); let i = index; let odd = odd;
            let even = even" [class.bg-info]="odd" [class.bg-warning]="even">
        <td style="vertical-align:middle">{{i + 1}}</td>
        <td style="vertical-align:middle">{{item.name}}</td>
        <td style="vertical-align:middle">{{item.category}}</td>
        <td style="vertical-align:middle">{{item.price | currency:"USD":true }}</td>
        <td class="text-xs-center">
            <button class="btn btn-danger btn-sm" (click)="deleteProduct(item.id)">
                Delete
            </button>
        </td>
    </tr>
</table>
```

Синтаксис применения канала напоминает синтаксис каналов командной строки: значение «передается» для преобразования символом | (вертикальная черта). На рис. 18.2 показана структура привязки данных с каналом.

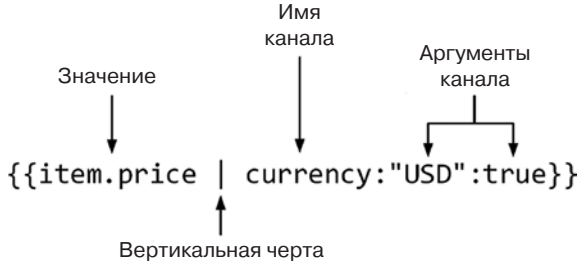


Рис. 18.2. Анатомия привязки данных с каналом

В листинге 18.8 используется имя канала `currency`. Этот канал преобразует числа в формат денежных сумм. Аргументы каналов разделяются двоеточиями (символ `:`). Первый аргумент канала задает код денежной единицы (в данном случае `USD` — доллары США). Второй аргумент канала, равный `true`, определяет, должен ли выводиться знак денежной единицы вместо ее кода.

Во время обработки выражения Angular получает значение данных и передает его каналу для преобразования. Результат, сгенерированный каналом, затем используется как результат выражения для привязки данных. В данном примере используются привязки со строковой интерполяцией, а результаты показаны на рис. 18.3.

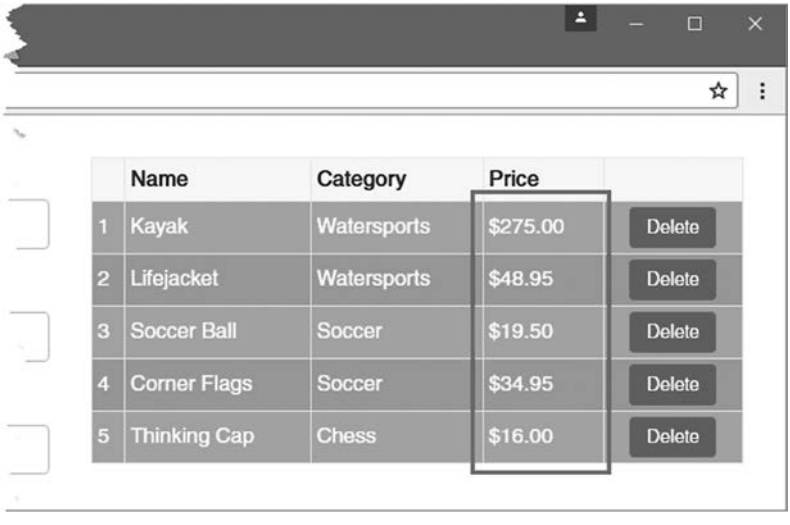


Рис. 18.3. Эффект использования канала

Создание нестандартного канала

Позднее в этой главе мы вернемся ко встроенным каналам, предоставляемым Angular. Лучший способ понять, как работают каналы и что они могут, — создать класс нестандартного канала. Создайте файл addTax.pipe.ts в папке app и включите в него определение класса из листинга 18.9.

Листинг 18.9. Содержимое файла addTax.pipe.ts в папке app

```
import { Pipe } from "@angular/core";

@Pipe({
    name: "addTax"
})
export class PaAddTaxPipe {

    defaultRate: number = 10;

    transform(value: any, rate?: any): number {
        let valueNumber = Number.parseFloat(value);
        let rateNumber = rate == undefined ?
            this.defaultRate : Number.parseInt(rate);
        return valueNumber + (valueNumber * (rateNumber / 100));
    }
}
```

Канал представляет собой класс с декоратором `@Pipe`, реализующий метод с именем `transform`. (Каналы также могут реализовать интерфейс `PipeTransform`, но это не обязательно, и в книге эта возможность не используется.) Декоратор `@Pipe` определяет два свойства, которые используются для настройки каналов (табл. 18.3).

Таблица 18.3. Свойства декоратора @Pipe

Имя	Описание
name	Свойство задает имя, под которым канал применяется к шаблонам
pure	Если значение свойства равно true, канал повторно обрабатывается только при изменении входного значения или его аргументов. Этот режим используется по умолчанию. За дополнительной информацией обращайтесь к разделу «Создание нечистых каналов»

Канал определяется в классе с именем `PaAddTaxPipe`, а свойство декоратора `name` указывает, что канал будет применяться с использованием `addTax` в шаблонах.

Метод `transform` должен получать как минимум один аргумент, который используется Angular для предоставления данных, с которыми работает канал. Канал выполняет свою работу в методе `transform`, а его результат используется Angular в выражении привязки. В этом примере канал получает значение `number` и вычисляет результат суммированием полученного значения и налога с продаж.

Метод `transform` также может определять дополнительный аргумент, который используется для настройки канала. В примере необязательный аргумент `rate` может использоваться для передачи ставки налога с продаж, который по умолчанию составляет 10%.

ВНИМАНИЕ

Будьте внимательны при работе с аргументами, передаваемыми методу transform. Проследите за тем, чтобы они были правильно разобраны или преобразованы к нужным типам. Аннотации типов TypeScript не проверяются на стадии выполнения, и Angular просто передаст используемые значения.

Регистрация нестандартного канала

Каналы регистрируются в свойстве `declarations` модуля Angular (листинг 18.10).

Листинг 18.10. Регистрация нестандартного канала в файле app.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { ProductComponent } from "./component";
import { FormsModule, ReactiveFormsModule  } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
import { PaModel } from "./twoway.directive";
import { PaStructureDirective } from "./structure.directive";
import { PaIteratorDirective } from "./iterator.directive";
import { PaCellColor } from "./cellColor.directive";
```

```
import { PaCellColorSwitcher } from "./cellColorSwitcher.directive";
import { ProductTableComponent } from "./productTable.component";
import { ProductFormComponent } from "./productForm.component";
import { PaToggleView } from "./toggleView.component";
import { PaAddTaxPipe } from "./addTax.pipe";

@NgModule({
    imports: [BrowserModule, FormsModule, ReactiveFormsModule],
    declarations: [ProductComponent, PaAttrDirective, PaModel,
        PaStructureDirective, PaIteratorDirective,
        PaCellColor, PaCellColorSwitcher, ProductTableComponent,
        ProductFormComponent, PaToggleView, PaAddTaxPipe],
    bootstrap: [ProductComponent]
})
export class AppModule { }
```

Применение нестандартного канала

После того как нестандартный канал будет зарегистрирован, вы сможете использовать его в выражениях привязки данных. В листинге 18.11 канал применяется к значению `price` в таблицах, а добавленный элемент `select` позволяет задать ставку налога.

Листинг 18.11. Применение нестандартного канала в файле productTable.component.html

```
<div>
    <label>Tax Rate:</label>
    <select [value]="taxRate || 0" (change)="taxRate=$event.target.value">
        <option value="0">None</option>
        <option value="10">10%</option>
        <option value="20">20%</option>
        <option value="50">50%</option>
    </select>
</div>

<table class="table table-sm table-bordered table-striped">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th><th></th></tr>
    <tr *paFor="let item of getProducts(); let i = index; let odd = odd;
            let even = even" [class.bg-info]="odd" [class.bg-warning]="even">
        <td style="vertical-align:middle">{{i + 1}}</td>
        <td style="vertical-align:middle">{{item.name}}</td>
        <td style="vertical-align:middle">{{item.category}}</td>
        <td style="vertical-align:middle">
            {{item.price | addTax:(taxRate || 0) }}
        </td>
        <td class="text-xs-center">
            <button class="btn btn-danger btn-sm" (click)="deleteProduct(item.id)">
                Delete
            </button>
        </td>
    </tr>
</table>
```

Просто для разнообразия я определяю ставку налога в шаблоне. Элемент `select` содержит привязку, которая задает его свойству `value` переменную компонента с именем `taxRate` (или значение по умолчанию 0, если свойство не определено). Привязка события обрабатывает событие `change` и задает значение свойства `taxRate`. При использовании директивы `ngModel` невозможно задать резервное значение, поэтому я и разбил привязки.

Применяя нестандартный канал, я использую символ вертикальной черты |, за которым следует значение, заданное свойством `name` в декораторе канала. За именем канала следует двоеточие, за ним следует выражение, результат которого предоставляет аргумент канала. В этом случае будет использоваться свойство `taxRate`, если оно определено, или резервное значение 0.

Каналы являются частью динамической природы привязок данных Angular, а метод `transform` канала будет вызываться для получения обновленного значения в случае изменения связанного значения данных или выражения, использованного для аргументов. Чтобы увидеть динамическую природу каналов, измените значение, отображаемое элементом `select`; это приведет к определению или изменению свойства `taxRate`, которое, в свою очередь, обновит величину, прибавляемую к свойству `price` из-за нестандартного канала (рис. 18.4).

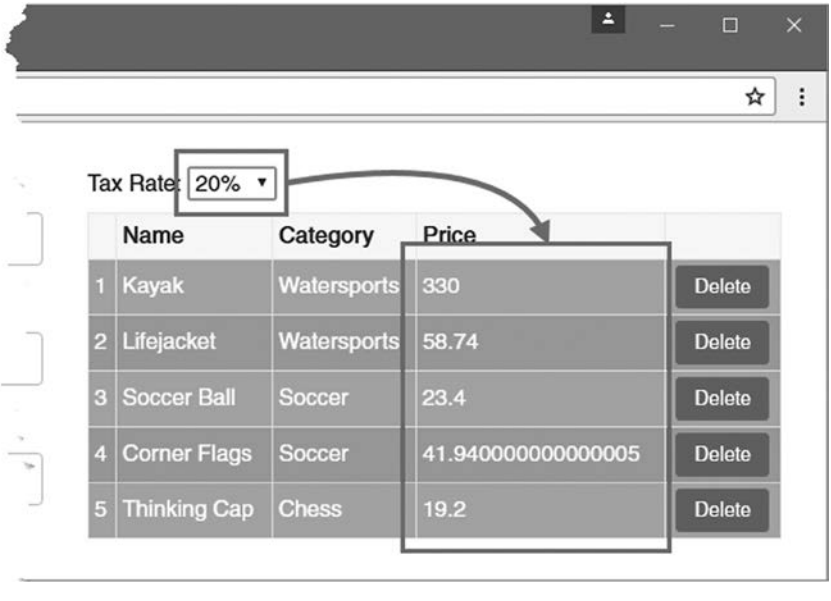


Рис. 18.4. Использование нестандартного канала

Объединение каналов

Канал `addTax` применяет ставку налога, но дробные величины, полученные в результате вычисления, выглядят некрасиво — и бесполезно, потому что налоговые службы обычно не настаивают на точности до 15 цифр в дробной части.

Проблему можно было бы решить, добавив в нестандартный канал поддержку форматирования числовых значений в денежные суммы, но для этого пришлось бы дублировать функциональность встроенного канала `currency`, который использовался ранее в этой главе.

Другое, более правильное решение — объединение функциональности обоих каналов, чтобы вывод нестандартного канала `addTax` подавался на встроенный канал `currency`, который выдает значение, отображаемое для пользователя. Каналы объединяются символом |, а их имена следуют в порядке передачи данных, как показано в листинге 18.12.

Листинг 18.12. Объявление каналов в файле productTable.component.html

```
...
<td style="vertical-align:middle">
    {{item.price | addTax:(taxRate || 0) | currency:"USD":true  }}
</td>
...
```

Значение свойства `item.price` передается каналу `addTax`, который прибавляет налог с продаж, а затем каналу `currency`, который форматирует значение `number` в денежную величину (рис. 18.5).

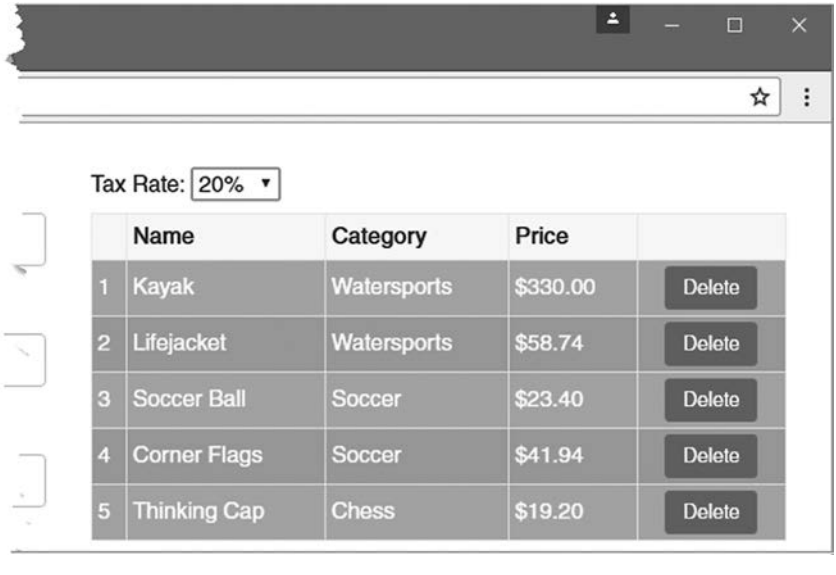


Рис. 18.5. Объединение функциональности каналов

Создание нечистых каналов

Свойство декоратора `pure` используется для того, чтобы сообщить Angular, когда следует вызывать метод `transform` канала. По умолчанию свойство `pure` равно

true; это сообщает Angular, что метод transform канала должен генерировать новое значение только при изменении входного значения данных (значение перед вертикальной чертой в шаблоне) или одного или нескольких из его аргументов. Такие каналы называются *чистыми* (pure), потому что у них нет независимого внутреннего состояния, и всеми их зависимостями можно управлять с использованием процесса обнаружения изменений в Angular.

Если задать свойству декоратора pure значение false, будет создан *нечистый канал*. Тем самым вы сообщаете Angular, что канал имеет собственные данные состояния или что он зависит от данных, которые не могут быть обнаружены в процессе поиска изменений при наличии нового значения.

В процессе обнаружения изменений Angular рассматривает нечистые каналы как самостоятельные источники значений и вызывает методы transform даже при отсутствии изменений в значениях данных и аргументах.

Нечистые каналы чаще всего используются при обработке содержимого массивов, элементы которых могут изменяться. Как было показано в главе 16, Angular не обнаруживает автоматически изменения, происходящие в массивах, и не вызывает метод transform чистого канала при редактировании или удалении элемента массива, потому что объект массива, используемый как входное значение данных, остается неизменным.

ВНИМАНИЕ

Нечистые каналы следует использовать осмотрительно, потому что Angular приходится вызывать метод transform при любых изменениях данных или взаимодействиях с пользователем — просто на случай, если это приведет к изменению результата другого канала. Если вы создаете нечистый канал, сделайте его по возможности простым. Выполнение сложных операций, например сортировка массива, может сильно повредить быстродействию приложения Angular.

Создайте файл categoryFilter.pipe.ts в папке app и включите в него определение канала из листинга 18.13.

Листинг 18.13. Содержимое файла categoryFilter.pipe.ts в папке app

```
import { Pipe } from "@angular/core";
import { Product } from "./product.model";

@Pipe({
    name: "filter",
    pure: true
})
export class PaCategoryFilterPipe {

    transform(products: Product[], category: string): Product[] {
        return category == undefined ?
            products : products.filter(p => p.category == category);
    }
}
```

Это чистый фильтр, который получает массив объектов `Product` и возвращает только те, у которых свойство `category` совпадает с аргументом `category`. В листинге 18.14 показано, как новый канал регистрируется в модуле Angular.

Листинг 18.14. Регистрация канала в файле app.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { ProductComponent } from "./component";
import { FormsModule, ReactiveFormsModule  } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
import { PaModel } from "./twoway.directive";
import { PaStructureDirective } from "./structure.directive";
import { PaIteratorDirective } from "./iterator.directive";
import { PaCellColor } from "./cellColor.directive";
import { PaCellColorSwitcher } from "./cellColorSwitcher.directive";
import { ProductTableComponent } from "./productTable.component";
import { ProductFormComponent } from "./productForm.component";
import { PaToggleView } from "./toggleView.component";
import { PaAddTaxPipe } from "./addTax.pipe";
import { PaCategoryFilterPipe } from "./categoryFilter.pipe";

@NgModule({
    imports: [BrowserModule, FormsModule, ReactiveFormsModule],
    declarations: [ProductComponent, PaAttrDirective, PaModel,
        PaStructureDirective, PaIteratorDirective,
        PaCellColor, PaCellColorSwitcher, ProductTableComponent,
        ProductFormComponent, PaToggleView, PaAddTaxPipe,
        PaCategoryFilterPipe],
    bootstrap: [ProductComponent]
})
export class AppModule { }
```

В листинге 18.15 продемонстрировано применение нового канала в выражении привязки, целью которой является директива `ngFor`, а также новый элемент `select` для выбора фильтра `category`.

Листинг 18.15. Применение канала в файле productTable.component.html

```
<div>
    <label>Tax Rate:</label>
    <select [value]="taxRate || 0" (change)="taxRate=$event.target.value">
        <option value="0">None</option>
        <option value="10">10%</option>
        <option value="20">20%</option>
        <option value="50">50%</option>
    </select>
</div>

<div>
    <label>Category Filter:</label>
    <select [(ngModel)]="categoryFilter">
```

```
        <option>Watersports</option>
        <option>Soccer</option>
        <option>Chess</option>
    </select>
</div>

<table class="table table-sm table-bordered table-striped">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th><th></th></tr>
    <tr *paFor="let item of getProducts() | filter:categoryFilter;
            let i = index; let odd = odd; let even = even"
            [class.bg-info]="odd" [class.bg-warning]="even">
        <td style="vertical-align:middle">{{i + 1}}</td>
        <td style="vertical-align:middle">{{item.name}}</td>
        <td style="vertical-align:middle">{{item.category}}</td>
        <td style="vertical-align:middle">
            {{item.price | addTax:(taxRate || 0) | currency:"USD":true  }}
        </td>
        <td class="text-xs-center">
            <button class="btn btn-danger btn-sm" (click)="deleteProduct(item.id)">
                Delete
            </button>
        </td>
    </tr>
</table>
```

Чтобы понять суть проблемы, используйте элемент `select` для фильтрации товаров в таблице, чтобы выводились только товары из категории `Soccer`. Затем используйте элементы `form` для создания нового товара в этой категории. Кнопка Create добавляет товар в модель данных, но новый товар не будет отображаться в таблице (рис. 18.6).

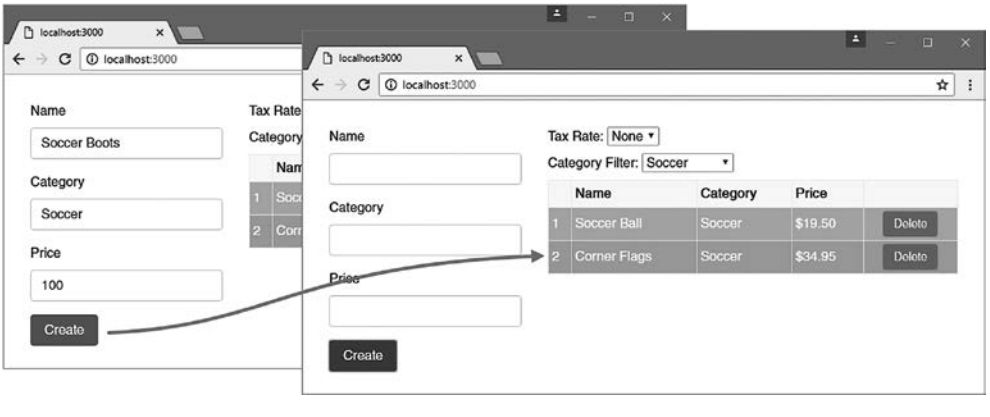


Рис. 18.6. Проблема, обусловленная использованием чистого канала

Таблица не обновляется, потому что с точки зрения Angular входные данные канала `filter` не изменились. Метод `getProducts` компонента возвращает тот же объ-

ект массива, а свойство `categoryFilter` по-прежнему содержит `Soccer`. Angular не замечает, что в массиве появился новый объект, который возвращается методом `getProducts`.

Проблема решается заданием свойству `pure` канала значения `false` (листинг 18.16).

Листинг 18.16. Пометка нечистого канала в файле categoryFilter.pipe.ts

```
import { Pipe } from "@angular/core";
import { Product } from "./product.model";

@Pipe({
    name: "filter",
    pure: false
})
export class PaCategoryFilterPipe {

    transform(products: Product[], category: string): Product[] {
        return category == undefined ?
            products : products.filter(p => p.category == category);
    }
}
```

Повторив тест, вы увидите, что новый товар корректно отображается в таблице (рис. 18.7).

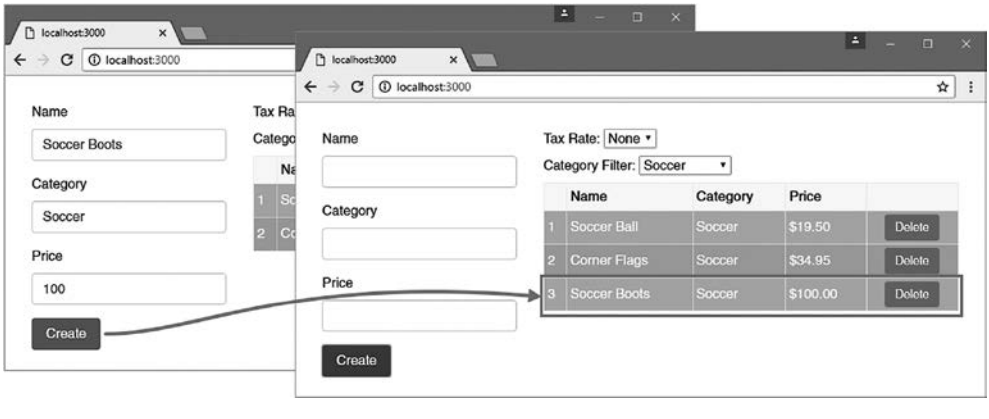


Рис. 18.7. Использование нечистых каналов

Использование встроенных каналов

Angular включает подборку встроенных каналов для выполнения часто встречающихся задач. Эти каналы подробно описаны в табл. 18.4, а их использование продемонстрировано ниже.

Таблица 18.4. Встроенные каналы

Имя	Описание
number	Канал форматирует числовые значения с учетом локального контекста. За дополнительной информацией обращайтесь к разделу «Форматирование чисел»
currency	Канал форматирует денежные суммы с учетом локального контекста. За дополнительной информацией обращайтесь к разделу «Форматирование денежных величин»
percent	Канал форматирует проценты с учетом локального контекста. За дополнительной информацией обращайтесь к разделу «Форматирование процентов»
date	Канал форматирует дату с учетом локального контекста. За дополнительной информацией обращайтесь к разделу «Форматирование дат»
uppercase	Канал преобразует все символы в строке к верхнему регистру. За дополнительной информацией обращайтесь к разделу «Изменение регистра символов»
lowercase	Канал преобразует все символы в строке к нижнему регистру. За дополнительной информацией обращайтесь к разделу «Изменение регистра символов»
json	Канал преобразует объект в строку в формате JSON. За дополнительной информацией обращайтесь к разделу «Сериализация данных в формате JSON»
slice	Канал выбирает элементы из массива или символы из строки. За дополнительной информацией обращайтесь к разделу «Срезы массивов данных»
async	Канал подписывается на Observable или Promise и выводит последнее из полученных значений. Использование каналов продемонстрировано в главе 23

ПРИМЕЧАНИЕ

Angular также включает каналы i18nPlural и i18nSelect, используемые для локализации контента. Поддержка локализации в Angular в этой книге не рассматривается, потому что она еще не готова для реальной эксплуатации. За подробностями обращайтесь к разделу http://angular.io/docs/ts/latest/cookbook/i18n.html.

Форматирование чисел

Канал `pipe` форматирует числа с учетом локального контекста; эта задача требует API интернационализации, для которого в начале этой главы было добавлено полизаполнение. В листинге 18.17 продемонстрировано использование канала `number` и аргумента, определяющего способ форматирования. Из шаблона были удалены нестандартные каналы и сопутствующие элементы `select`.

Листинг 18.17. Использование канала number в файле productTable.component.html

```
<table class="table table-sm table-bordered table-striped">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th><th></th></tr>
    <tr *paFor="let item of getProducts(); let i = index; let odd = odd;
            let even = even" [class.bg-info]="odd" [class.bg-warning]="even">
        <td style="vertical-align:middle">{{i + 1}}</td>
        <td style="vertical-align:middle">{{item.name}}</td>
        <td style="vertical-align:middle">{{item.category}}</td>
        <td style="vertical-align:middle">{{item.price | number:"3.2-2" }}</td>
        <td class="text-xs-center">
            <button class="btn btn-danger btn-sm" (click)="deleteProduct(item.id)">
                Delete
            </button>
        </td>
    </tr>
</table>
```

Канал `number` получает один аргумент, который задает количество цифр в отформатированном результате. Аргумент имеет следующий формат (обратите внимание на точку и дефис, разделяющие значения, а весь аргумент заключается в кавычки как строка):

```
"<minIntegerDigits>.<minFactionDigits>-<maxFractionDigits>"
```

Все элементы аргумента `formatting` описаны в табл. 18.5.

Таблица 18.5. Элементы аргумента канала number

Имя	Описание
minIntegerDigits	Минимальное количество цифр в целой части (по умолчанию 1)
minFractionDigits	Минимальное количество цифр в дробной части (по умолчанию 0)
maxFractionDigits	Максимальное количество цифр в дробной части (по умолчанию 3)

В листинге используется аргумент `"3.2-2"`; это означает, что для вывода целой части числа следует использовать не менее трех цифр, а дробная часть должна содержать не менее двух цифр. Результат показан на рис. 18.8.

Канал `number` учитывает локальный контекст; это означает, что с одним и тем же аргументом формата будут получены разные результаты в зависимости от настроек локального контекста пользователя. Приложения Angular по умолчанию используют локальный контекст `en-US`; смена контекста должна осуществляться явно, при помощи записи в корневом модуле (листинг 18.18).

Листинг 18.18. Настройка локального контекста в файле app.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { ProductComponent } from "./component";
import { FormsModule, ReactiveFormsModule } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
```

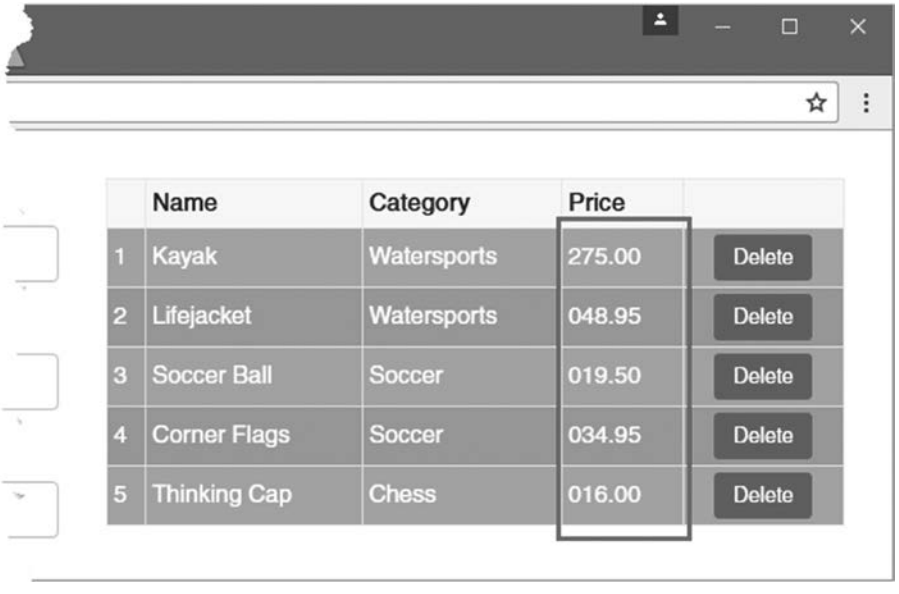


Рис. 18.8. Форматирование числовых значений

```
import { PaModel } from "./twoway.directive";
import { PaStructureDirective } from "./structure.directive";
import { PaIteratorDirective } from "./iterator.directive";
import { PaCellColor } from "./cellColor.directive";
import { PaCellColorSwitcher } from "./cellColorSwitcher.directive";
import { ProductTableComponent } from "./productTable.component";
import { ProductFormComponent } from "./productForm.component";
import { PaToggleView } from "./toggleView.component";
import { PaAddTaxPipe } from "./addTax.pipe";
import { PaCategoryFilterPipe } from "./categoryFilter.pipe";
import { LOCALE_ID } from "@angular/core";

@NgModule({
    imports: [BrowserModule, FormsModule, ReactiveFormsModule],
    declarations: [ProductComponent, PaAttrDirective, PaModel,
        PaStructureDirective, PaIteratorDirective,
        PaCellColor, PaCellColorSwitcher, ProductTableComponent,
        ProductFormComponent, PaToggleView, PaAddTaxPipe,
        PaCategoryFilterPipe],
    providers: [{ provide: LOCALE_ID, useValue: "fr-FR" }],
    bootstrap: [ProductComponent]
})
export class AppModule { }
```

Настройка локального контекста зависит от функциональности, описанной в главе 20, но, по сути, здесь происходит переключение на локальный контекст `fr-FR` для французского языка, используемого во Франции; это приводит к изменению формата числовых значений (рис. 18.9).

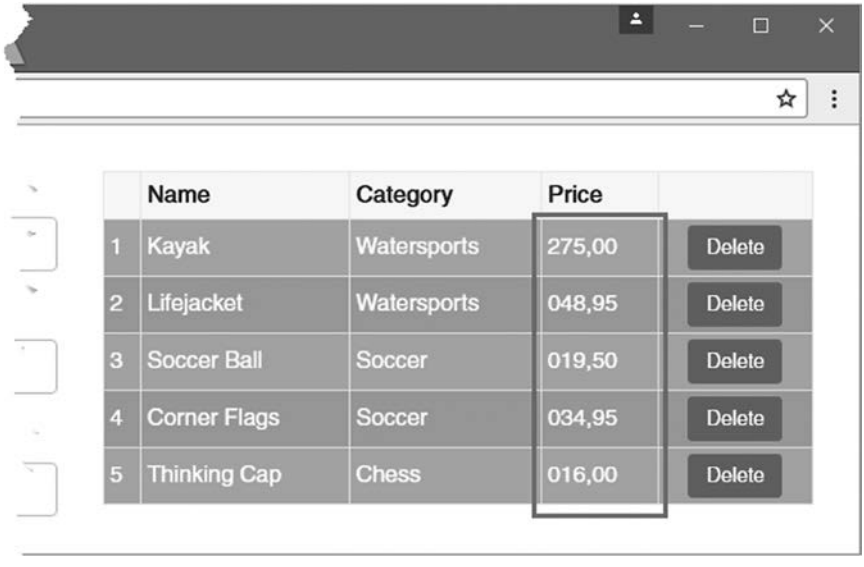


Рис. 18.9. Форматирование с учетом локального контекста

Форматирование денежных величин

Канал `currency` форматирует числовые значения, представляющие денежные величины. В листинге 18.18 этот канал используется для демонстрации, а в листинге 18.19 показано применение того же канала с добавлением спецификаторов числового форматирования.

Листинг 18.9. Использование канала currency в файле productTable.component.html

```
<table class="table table-sm table-bordered table-striped">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th><th></th></tr>
    <tr *paFor="let item of getProducts(); let i = index; let odd = odd;
            let even = even" [class.bg-info]="odd" [class.bg-warning]="even">
        <td style="vertical-align:middle">{{i + 1}}</td>
        <td style="vertical-align:middle">{{item.name}}</td>
        <td style="vertical-align:middle">{{item.category}}</td>
        <td style="vertical-align:middle">
            {{item.price | currency:"USD":true:"2.2-2" }}
        </td>
        <td class="text-xs-center">
            <button class="btn btn-danger btn-sm" (click)="deleteProduct(item.id)">
                Delete
            </button>
        </td>
    </tr>
</table>
```

Для настройки канала `currency` используются три аргумента, описанные в табл. 18.6.

Таблица 18.6. Аргументы канала currency

Имя	Описание
currencyCode	Строковый аргумент задает денежную единицу в виде кода ISO 4217. По умолчанию при отсутствии аргумента используется значение USD. Список кодов денежных единиц доступен по адресу http://en.wikipedia.org/wiki/ISO_4217
symbolDisplay	Если значение равно true, этот аргумент Boolean указывает, что выводиться должен знак денежной единицы. Со значением false будет использоваться код денежной единицы ISO 4217. По умолчанию используется значение false
digitInfo	Строковый аргумент задает форматирование числа с использованием инструкций форматирования, поддерживаемых каналом number (см. раздел «Форматирование чисел»)

Аргументы в листинге 18.19 приказывают каналу использовать при выводе знак доллара США (код ISO `USD`) вместо кода и форматировать число так, чтобы оно содержало не менее двух цифр в целой части и ровно две цифры в дробной части.

Канал использует API интернационализации для получения информации о денежной единице (прежде всего знаке), но не выбирает денежную единицу автоматически в соответствии с настройками локального контекста пользователя.

Это означает, что форматирование числа и позиция денежного знака зависят от локального контекста приложения независимо от того, какая денежная единица была задана каналом. Наш пример по-прежнему настроен для использования локального контекста `fr-FR`, в результате чего будет получен результат, показанный на рис. 18.10.

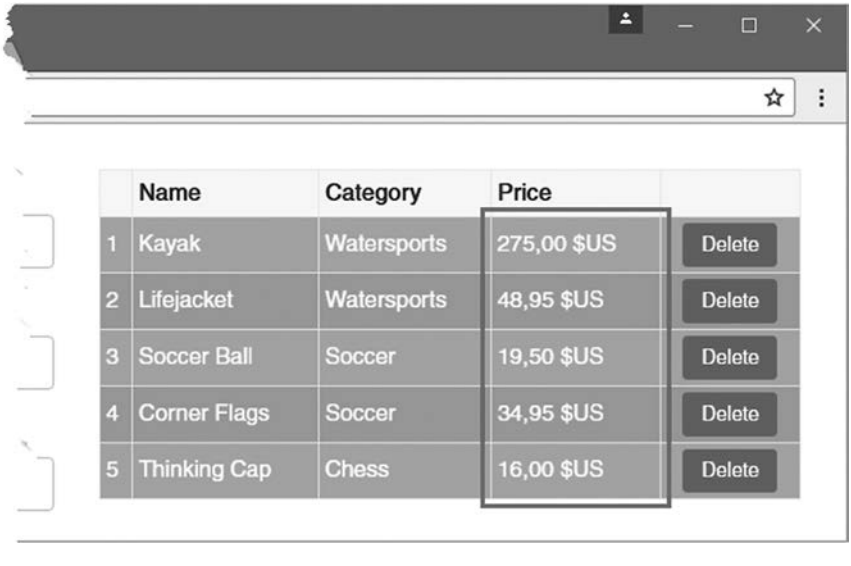


Рис. 18.10. Форматирование денежных сумм с учетом локального контекста

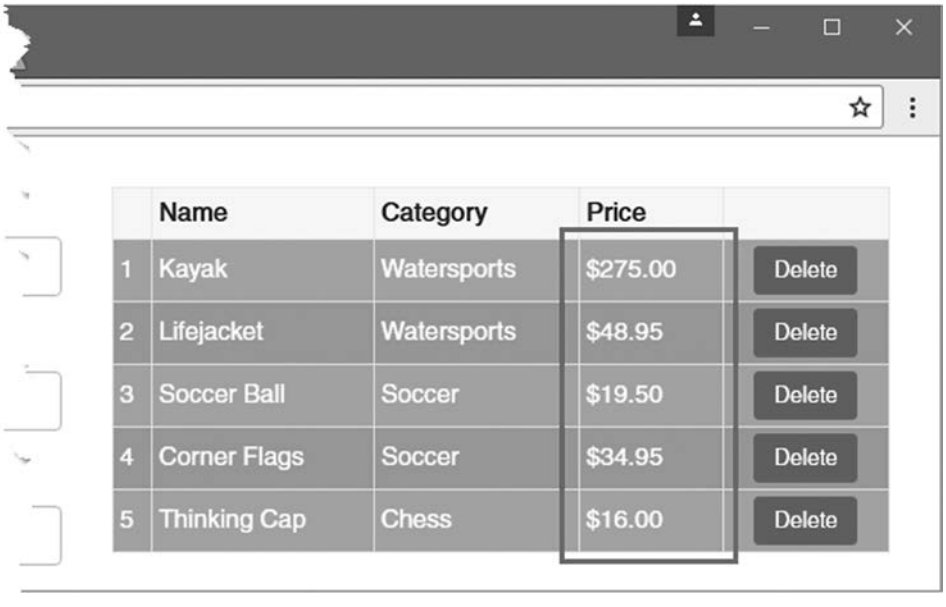


Рис. 18.11. Форматирование денежных величин

Чтобы вернуться к локальному контексту по умолчанию, в листинге 18.20 запись `fr-FR` удаляется из корневого модуля приложения.

Листинг 18.20. Удаление локального контекста из файла app.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { ProductComponent } from "./component";
import { FormsModule, ReactiveFormsModule } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
import { PaModel } from "./twoway.directive";
import { PaStructureDirective } from "./structure.directive";
import { PaIteratorDirective } from "./iterator.directive";
import { PaCellColor } from "./cellColor.directive";
import { PaCellColorSwitcher } from "./cellColorSwitcher.directive";
import { ProductTableComponent } from "./productTable.component";
import { ProductFormComponent } from "./productForm.component";
import { PaToggleView } from "./toggleView.component";
import { PaAddTaxPipe } from "./addTax.pipe";
import { PaCategoryFilterPipe } from "./categoryFilter.pipe";
//import { LOCALE_ID } from "@angular/core";

@NgModule({
    imports: [BrowserModule, FormsModule, ReactiveFormsModule],
    declarations: [ProductComponent, PaAttrDirective, PaModel,
        PaStructureDirective, PaIteratorDirective,
        PaCellColor, PaCellColorSwitcher, ProductTableComponent,
        ProductFormComponent, PaToggleView, PaAddTaxPipe,
```

```
        PaCategoryFilterPipe],
    //providers: [{ provide: LOCALE_ID, useValue: "fr-FR" }],
    bootstrap: [ProductComponent]
})
export class AppModule { }
```

Результат показан на рис. 18.11.

Форматирование процентов

Канал `percent` форматирует числовые значения в процентах; значения в диапазоне от 0 до 1 форматируются так, чтобы они представляли от 0 до 100 процентов. Канал имеет необязательный аргумент, который используется для задания параметров числового форматирования в таком же формате, как у канала `number`. Листинг 18.21 возвращает фильтр прибавления налога и заполняет ассоциированный элемент `select` элементами `option`, контент которых форматируется с применением фильтра `percent`.

Листинг 18.21. Форматирование процентов в файле productTable.component.html

```
<div>
    <label>Tax Rate:</label>
    <select [value]="taxRate || 0" (change)="taxRate=$event.target.value">
        <option value="0">None</option>
        <option value="10">{{ 0.1 | percent }}</option>
        <option value="20">{{ 0.2 | percent }}</option>
        <option value="50">{{ 0.5 | percent }}</option>
        <option value="150">{{ 1.5 | percent }}</option>
    </select>
</div>

<table class="table table-sm table-bordered table-striped">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th><th></th></tr>
    <tr *paFor="let item of getProducts(); let i = index; let odd = odd;
            let even = even" [class.bg-info]="odd" [class.bg-warning]="even">
        <td style="vertical-align:middle">{{i + 1}}</td>
        <td style="vertical-align:middle">{{item.name}}</td>
        <td style="vertical-align:middle">{{item.category}}</td>
        <td style="vertical-align:middle">
            {{item.price | addTax:(taxRate || 0) | currency:"USD":true:"2.2-2" }}
        </td>
        <td class="text-xs-center">
            <button class="btn btn-danger btn-sm" (click)="deleteProduct(item.id)">
                Delete
            </button>
        </td>
    </tr>
</table>
```

Значения, большие 1, форматируются в значения более 100%. Пример приведен на рис. 18.12, где для значения 1.5 генерируется отформатированное значение `150%`.

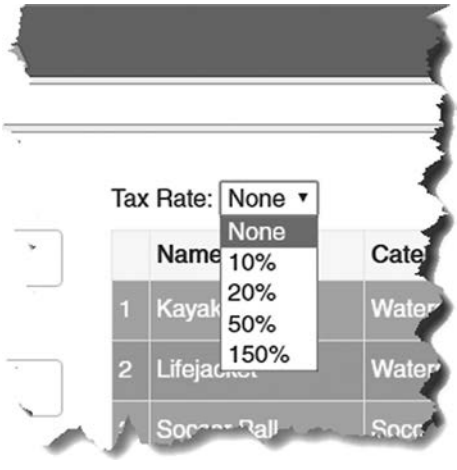


Рис. 18.12. Форматирование процентов

Форматирование процентов выполняется с учетом локального контекста, хотя различия между локальными контекстами порой бывают нетривиальными. Например, если в локальном контексте `en-US` получается результат 10%, где число и знак процента находятся рядом друг с другом, многие другие контексты, включая `fr-FR`, отделяют число от знака процента пробелом.

Форматирование дат

Канал данных форматирует даты с учетом локального контекста. Даты могут выражаться объектами JavaScript `Date`, в виде числовых значений, представляющих количество миллисекунд с начала 1970 года, или в виде строки со строго определенным форматом. В листинге 18.22 в класс `ProductTableComponent` добавляются три свойства, каждое из которых кодирует дату в одном из форматов, поддерживаемых каналом `date`.

Листинг 18.22. Определение дат в файле productTable.component.ts

```
import { Component, Input, ViewChildren, QueryList } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";
@Component({
    selector: "paProductTable",
    templateUrl: "app/productTable.component.html"
})
export class ProductTableComponent {

    @Input("model")
    dataModel: Model;
```

```
    getProduct(key: number): Product {
        return this.dataModel.getProduct(key);
    }

    getProducts(): Product[] {
        return this.dataModel.getProducts();
    }

    deleteProduct(key: number) {
        this.dataModel.deleteProduct(key);
    }

    dateObject: Date = new Date(2020, 1, 20);
    dateString: string = "2020-02-20T00:00:00.000Z";
    dateNumber: number = 1582156800000;
}
```

Все три свойства описывают одну и ту же дату: 20 февраля 2020 года без указания времени. В листинге 18.23 канал `date` используется для форматирования всех трех свойств.

Листинг 18.23. Форматирование дат в файле productTable.component.html

```
<div class="bg-info m-a-1 p-a-1">
    <div>Date formatted from object: {{ dateObject | date }}</div>
    <div>Date formatted from string: {{ dateString | date }}</div>
    <div>Date formatted from number: {{ dateNumber | date }}</div>
</div>

<table class="table table-sm table-bordered table-striped">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th><th></th></tr>
    <tr *paFor="let item of getProducts(); let i = index; let odd = odd;
            let even = even" [class.bg-info]="odd" [class.bg-warning]="even">
        <td style="vertical-align:middle">{{i + 1}}</td>
        <td style="vertical-align:middle">{{item.name}}</td>
        <td style="vertical-align:middle">{{item.category}}</td>
        <td style="vertical-align:middle">
            {{item.price | addTax:(taxRate || 0) | currency:"USD":true:"2.2-2" }}
        </td>
        <td class="text-xs-center">
            <button class="btn btn-danger btn-sm" (click)="deleteProduct(item.id)">
                Delete
            </button>
        </td>
    </tr>
</table>
```

Канал определяет, с каким типом данных он работает, разбирает значение для получения даты и форматирует его (рис. 18.13).

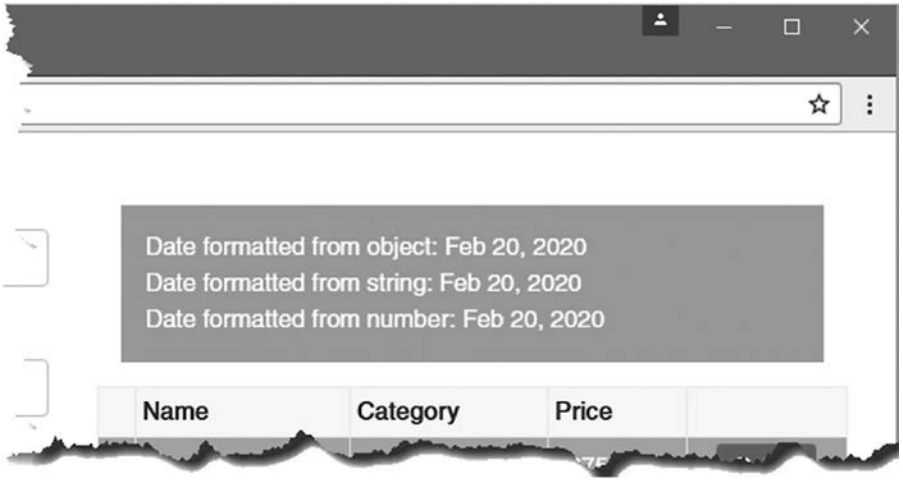


Рис. 18.13. Форматирование дат

Канал `date` получает один аргумент, который задает формат даты. Для выбора выводимых компонентов даты могут использоваться метасимволы из табл. 18.7.

Таблица 18.7. Символы формата канала date

Символ	Описание
y, yy	Год
M, MMM, MMMM	Месяц
d, dd	День (в числовом представлении)
E, EE, EEEE	День (в текстовом виде)
j, jj	Час
h, hh, H, HH	Час в 12- и 24-часовом формате
m, mm	Минуты
s, ss	Секунды
Z	Часовой пояс

Символы из табл. 18.7 предоставляют доступ к компонентам даты на разных уровнях детализации: так, для февраля `M` вернет 2, `MM` — 02, MMM — `Feb`, а `MMMM` — `February` (при использовании локального контекста `en-US`). Канал `date` также поддерживает предопределенные форматы даты для часто используемых комбинаций (табл. 18.8).

Таблица 18.8. Предопределенные форматы канала date

Имя	Описание
short	Формат эквивалентен строке yMdjm: дата представляется в компактном формате с включением компонента времени
medium	Формат эквивалентен строке yMMMdjms: дата представляется в расширенном формате с включением компонента времени
shortDate	Формат эквивалентен строке yMd: дата представляется в компактном формате без компонента времени
mediumDate	Формат эквивалентен строке yMMMd: дата представляется в расширенном формате без компонента времени
longDate	Формат эквивалентен строке yMMMMd: дата представляется без компонента времени
fullDate	Формат эквивалентен строке yMMMMEEEEd: дата представляется в полном формате без компонента даты
shortTime	Формат эквивалентен строке jm
mediumTime	Формат эквивалентен строке jms

В листинге 18.24 продемонстрировано использование предопределенных форматов в аргументах канала `date`, с выводом одной и той же даты в разных форматах.

Листинг 18.24. Форматирование дат в файле productTable.component.html

```
...
<div class="bg-info m-a-1 p-a-1">
    <div>Date formatted as shortDate: {{ dateObject | date:"shortDate" }}</div>
    <div>Date formatted as mediumDate: {{ dateObject | date:"mediumDate" }}</div>
    <div>Date formatted as longDate: {{ dateObject | date:"longDate" }}</div>
</div>
...
```

Аргументы задаются в виде строковых литералов. Будьте внимательны с регистром символов форматной строки: `shortDate` интерпретируется как один из предопределенных форматов из табл. 18.8, `shortdate` (с буквой `d` в нижнем регистре) будет интерпретироваться как последовательность символов из табл. 18.7, а результат будет бессмысленным.

ВНИМАНИЕ

Разбор и форматирование даты — процесс сложный и долгий. По этой причине свойство pure для канала date равно true; в результате изменения отдельных компонентов объекта Date не будут инициировать обновление. Если вы хотите, чтобы в выводимой дате отражались изменения, вы должны изменить ссылку на объект Date, на который ссылается привязка, содержащая канал date.

Форматирование даты осуществляется с учетом локального контекста; это означает, что вы будете получать разные компоненты даты для разных локальных контекстов. Не следует предполагать, что формат даты, имеющий смысл в одном локальном контексте, будет содержательным в другом. На рис. 18.14 показаны отформатированные данные в локальных контекстах `en-US` и `fr-FR`.

Рис. 18.14. Форматирование даты с учетом локального контекста

Изменение регистра символов в строке

Каналы `uppercase` и `lowercase` преобразуют все символы в строке к верхнему или нижнему регистру соответственно. В листинге 18.25 продемонстрировано применение обоих каналов к ячейкам таблицы `product`.

Листинг 18.25. Изменение регистра символов в файле productTable.component.html

```
<table class="table table-sm table-bordered table-striped">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th><th></th></tr>
    <tr *paFor="let item of getProducts(); let i = index; let odd = odd;
            let even = even" [class.bg-info]="odd" [class.bg-warning]="even">
        <td style="vertical-align:middle">{{i + 1}}</td>
        <td style="vertical-align:middle">{{item.name | uppercase }}</td>
        <td style="vertical-align:middle">{{item.category | lowercase }}</td>
        <td style="vertical-align:middle">
            {{item.price | addTax:(taxRate || 0) | currency:"USD":true:"2.2-2" }}
        </td>
        <td class="text-xs-center">
            <button class="btn btn-danger btn-sm" (click)="deleteProduct(item.id)">
                Delete
            </button>
        </td>
    </tr>
</table>
```

Эти каналы используют стандартные строковые методы JavaScript `toUpperCase` и `toLowerCase`, которые не учитывают настройки локального контекста (рис. 18.15).

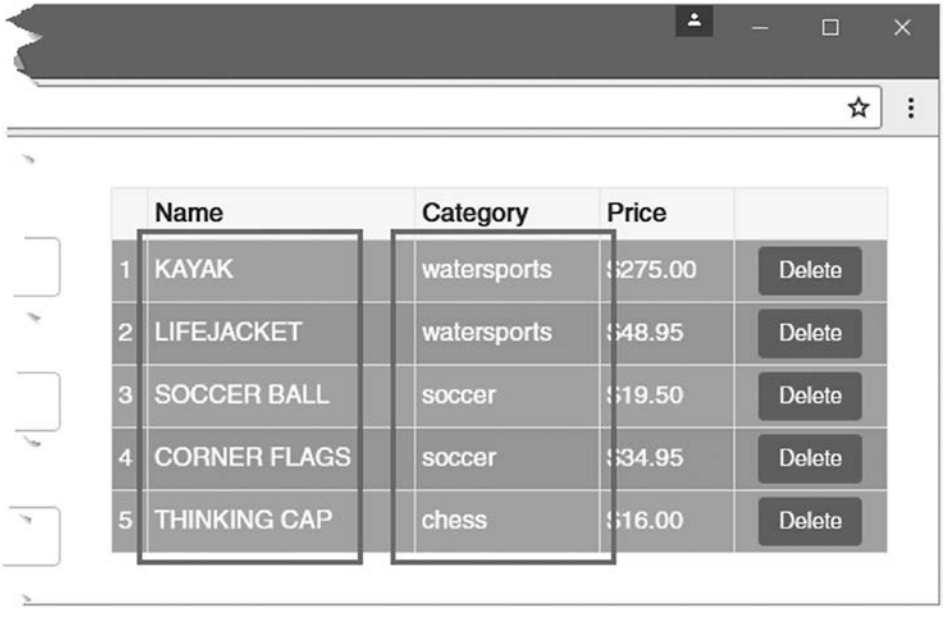


Рис. 18.15. Изменение регистра символов

Сериализация данных в формате JSON

Канал `json` создает представление значения данных в формате JSON. Никакие аргументы этому каналу не передаются. Для создания строки JSON используется метод `JSON.stringify` браузера. В листинге 18.26 канал используется для создания представления объектов модели данных в формате JSON.

Листинг 18.26. Создание строки JSON в файле productTable.component.html

```
<div class="bg-info m-a-1 p-a-1">
    <div>{{ getProducts() | json }}</div>
</div>

<table class="table table-sm table-bordered table-striped">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th><th></th></tr>
    <tr *paFor="let item of getProducts(); let i = index; let odd = odd;
            let even = even" [class.bg-info]="odd" [class.bg-warning]="even">
        <td style="vertical-align:middle">{{i + 1}}</td>
        <td style="vertical-align:middle">{{item.name | uppercase }}</td>
        <td style="vertical-align:middle">{{item.category | lowercase }}</td>
        <td style="vertical-align:middle">
            {{item.price | addTax:(taxRate || 0) | currency:"USD":true:"2.2-2" }}
        </td>
        <td class="text-xs-center">
            <button class="btn btn-danger btn-sm" (click)="deleteProduct(item.id)">
```

```
                Delete
            </button>
        </td>
    </tr>
</table>
```

Этот канал часто используется во время отладки. Свойство `pure` декоратора равно `false`, так что любое изменение в приложении приводит к вызову метода `transform` канала; это гарантирует, что будут отображаться даже изменения уровня коллекции. На рис. 18.16 показана разметка JSON, сгенерированная для объектов модели данных нашего примера.

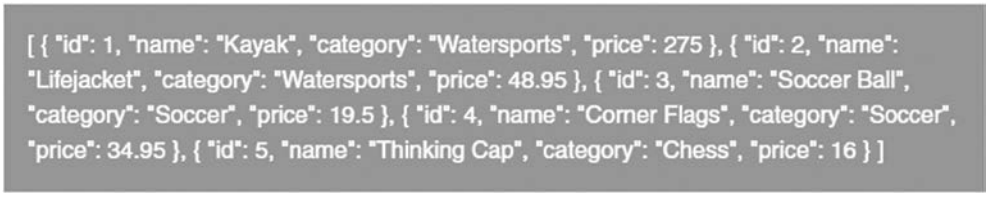


Рис. 18.16. Генерирование строк JSON для отладки

Срезы массивов данных

Канал `slice` берет массив или строку и возвращает подмножество элементов (или символов). Канал является нечистым; это означает, что он будет отражать любые изменения, происходящие с объектом данных, но с другой стороны, выборка подмножества будет выполняться после любого изменения в приложении, даже не связанного с данными источника.

Объекты или символы, выбранные каналом `slice`, задаются двумя аргументами, описанными в табл. 18.9.

Таблица 18.9. Аргументы канала currency

Имя	Описание
start	Этот аргумент является обязательным. Если его значение положительно, то начальный индекс элементов, включаемых в подмножество, отсчитывается от первого элемента массива. Если значение отрицательно, то канал отсчитывает индекс от конца массива
end	Необязательный аргумент указывает, сколько элементов, начиная с индекса start, должно быть включено в результат. Если значение не указано, то будут включены все элементы после индекса start (или до него, если значение отрицательно)

В листинге 18.27 продемонстрировано использование канала `slice` в сочетании с элементом `select`, которое указывает, сколько элементов должно отображаться в таблице товаров.

Листинг 18.27. Использование канала slice в файле productTable.component.html

```
<div>
    <label>Number of items:</label>
    <select [value]="itemCount || 1" (change)="itemCount=$event.target.value">
        <option *ngFor="let item of getProducts(); let i = index" [value]="i + 1">
            {{i + 1}}
        </option>
    </select>
</div>

<table class="table table-sm table-bordered table-striped">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th><th></th></tr>
    <tr *paFor="let item of getProducts() | slice:0:(itemCount || 1);
            let i = index; let odd = odd; let even = even"
            [class.bg-info]="odd" [class.bg-warning]="even">
        <td style="vertical-align:middle">{{i + 1}}</td>
        <td style="vertical-align:middle">{{item.name | uppercase }}</td>
        <td style="vertical-align:middle">{{item.category | lowercase }}</td>
        <td style="vertical-align:middle">
            {{item.price | addTax:(taxRate || 0) | currency:"USD":true:"2.2-2" }}
        </td>
        <td class="text-xs-center">
            <button class="btn btn-danger btn-sm" (click)="deleteProduct(item.id)">
                Delete
            </button>
        </td>
    </tr>
</table>
```

Элемент `select` заполняется элементами `option`, созданными директивой `ngFor`. Эта директива не позволяет напрямую выполнить заданное число итераций, поэтому я использовал переменную `index` для генерирования нужных значений. Элемент `select` задает значение переменной `itemCount`, которая используется как второй аргумент канала `slice`:

```
...
<tr *paFor="let item of getProducts() | slice:0:(itemCount || 1);
    let i = index; let odd = odd; let even = even"
    [class.bg-info]="odd" [class.bg-warning]="even">
...
```

В результате значение, отображаемое в элементе `select`, изменяет количество строк в таблице товаров (рис. 18.17).

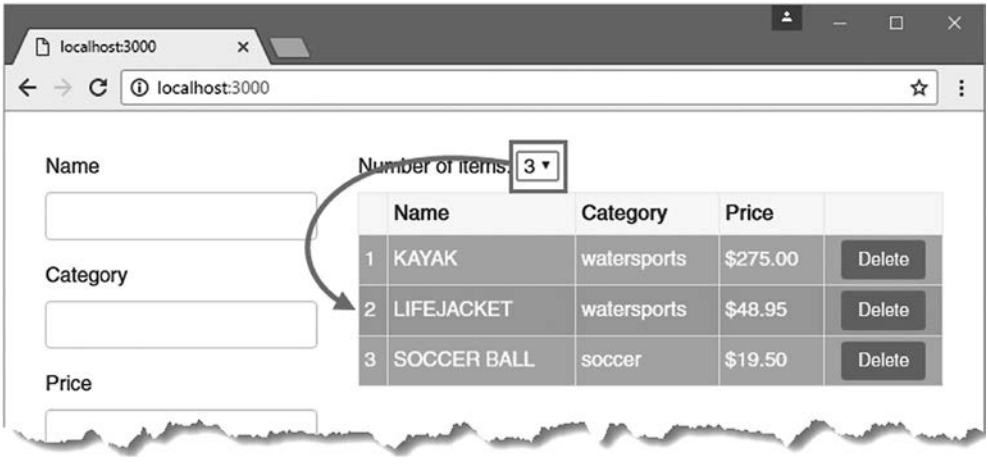


Рис. 18.17. Использование канала slice

Итоги

В этой главе рассматриваются каналы и их применение для преобразования данных, которые должны выводиться в шаблоне. Я продемонстрировал процесс создания нестандартных каналов, объяснил, чем чистые каналы отличаются от нечистых, и описал встроенные каналы, реализованные в Angular для выполнения типичных задач. Следующая глава посвящена службам, которые упрощают проектирование приложений Angular и позволяют легко организовать взаимодействие между структурными блоками.

19 Службы

Службы (services) — это объекты, предоставляющие общую функциональность для поддержки других структурных блоков приложения, таких как директивы, компоненты и каналы. Главной особенностью служб является механизм их использования через процесс, называемый внедрением зависимостей. Использование служб улучшает гибкость и масштабируемость приложений Angular, но понять, как работает внедрение зависимостей и какую пользу оно приносит, будет сложнее. Поэтому я начну эту главу с объяснения того, для решения каких проблем используются службы и внедрение зависимостей, как работает внедрение зависимостей и почему вам стоит применять службы в своих проектах. В главе 20 будут представлены дополнительные возможности, предоставляемые Angular для служб. В табл. 19.1 службы представлены в контексте.

Таблица 19.1. Каналы в контексте

Вопрос	Ответ
Что это такое?	Службы — объекты, определяющие функциональность, необходимую для других структурных блоков (таких, как компоненты или директивы). Службы отличаются от обычных объектов тем, что они предоставляются структурным блокам внешним провайдером, а не создаются напрямую ключевым словом new или передачей через входное свойство
Для чего они нужны?	Службы упрощают структуру приложений, перемещение и повторное использование функциональности и изоляцию структурных блоков в процессе модульного тестирования
Как они используются?	Классы объявляют зависимости от служб с использованием параметров конструктора, которые затем обрабатываются для набора служб, указанного в конфигурации приложения. Службы представляют собой классы, к которым применяется декоратор @Injectable
Есть ли у них недостатки или скрытые проблемы?	Внедрение зависимостей — достаточно спорная тема, и не все разработчики любят использовать этот механизм. Если вы не выполняете модульные тесты или ваши приложения относительно просты, то дополнительная работа, необходимая для реализации внедрения зависимостей, вряд ли окупится в долгосрочной перспективе
Есть ли альтернативы?	Вряд ли вам удастся обойтись без служб и внедрения зависимостей, потому что Angular использует их для предоставления доступа к своей встроенной функциональности. Однако вы не обязаны определять службы для своей функциональности, если не хотите

В табл. 19.2 приведена краткая сводка материала главы.

Таблица 19.2. Сводка материала главы

Проблема	Решение	Листинг
Предотвращение необходимости в ручном распределении совместно используемых объектов	Используйте службы	1–14, 21–28
Объявление зависимости от службы	Добавьте аргумент конструктора с типом службы	15–20

Подготовка проекта

В этой главе мы продолжим использовать проект example, который был создан в главе 11. Чтобы подготовить его для этой главы, удалите большую часть каналов из выражений привязки данных в таблице товаров, а также удалите элемент `select`, который использовался для выбора количества отображаемых товаров (листинг 19.1).

Листинг 19.1. Удаление канала из файла productTable.component.html

```
<table class="table table-sm table-bordered table-striped">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th><th></th></tr>
    <tr *paFor="let item of getProducts(); let i = index;
            let odd = odd; let even = even" [class.bg-info]="odd"
            [class.bg-warning]="even">
        <td style="vertical-align:middle">{{i + 1}}</td>
        <td style="vertical-align:middle">{{item.name}}</td>
        <td style="vertical-align:middle">{{item.category}}</td>
        <td style="vertical-align:middle">
            {{item.price | currency:"USD":true }}
        </td>
        <td class="text-xs-center">
            <button class="btn btn-danger btn-sm" (click)="deleteProduct(item.id)">
                Delete
            </button>
        </td>
    </tr>
</table>
```

Выполните следующую команду из папки example, чтобы запустить компилятор TypeScript и сервер HTTP для разработки:

```
npm start
```

Открывается новая вкладка браузера с контентом, показанным на рис. 19.1.

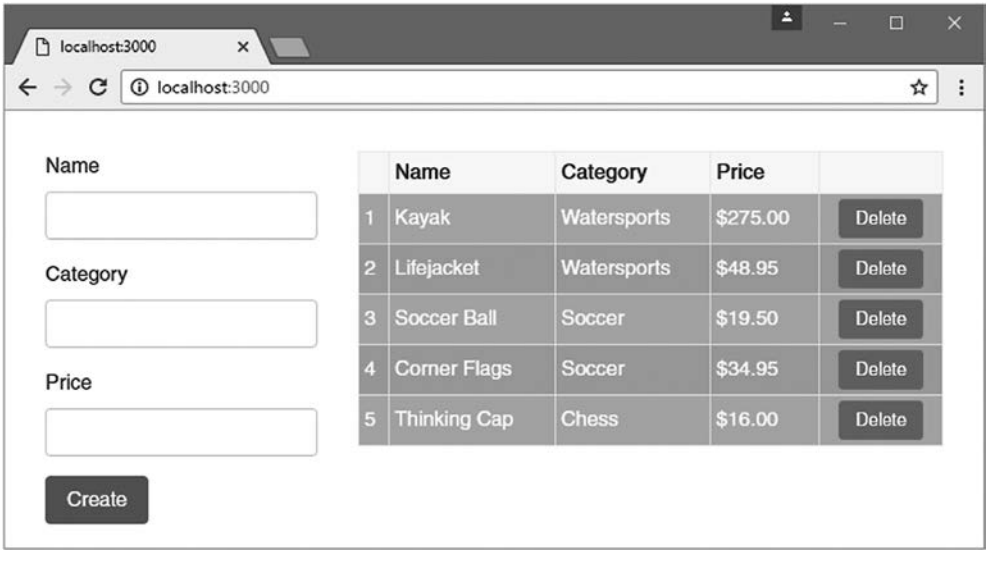


Рис. 19.1. Запуск приложения

Проблема распределения объектов

В главе 17 я добавил в проект компоненты, которые помогают избавиться от монолитной структуры приложения. При этом я использовал входные и выходные свойства для связывания компонентов, применив управляющие элементы для установления связи через изоляцию родительского компонента и его потомков, реализуемую Angular. Также было показано, как запрашивать контент шаблона для потомков представления; эта возможность дополняет функциональность контентных потомков, описанную в главе 16.

При хорошо продуманном применении эти механизмы координации между директивами и компонентами могут быть мощными и практичными. Но они также превращаются в обобщенный инструмент распределения совместно используемых объектов в приложениях, а это приводит к возрастанию сложности приложения и формированию жестких связей между компонентами.

Суть проблемы

Чтобы продемонстрировать суть проблемы, добавьте в проект совместно используемый объект и два компонента, которые от него зависят. Создайте файл discount.service.ts в папке app и определите в нем класс из листинга 19.2. Далее в этой главе я расскажу, почему важно, чтобы в имени файла присутствовала часть service.

Листинг 19.2. Содержимое файла discount.service.ts в папке app

```
export class DiscountService {
    private discountValue: number = 10;

    public get discount(): number {
        return this.discountValue;
    }

    public set discount(newValue: number) {
        this.discountValue = newValue || 0;
    }

    public applyDiscount(price: number) {
        return Math.max(price - this.discountValue, 5);
    }
}
```

Класс `DiscountService` определяет приватное свойство `discountValue`, которое используется для хранения величины скидки на товары в модели данных. Для работы с ним используются методы доступа, а также имеется вспомогательный метод `applyDiscount`, который сокращает цену товара при минимальной величине скидки 5.

Для первого компонента, использующего класс `DiscountService`, добавьте файл discountDisplay.component.ts в папку app и включите в него код из листинга 19.3.

Листинг 19.3. Содержимое файла discountDisplay.component.ts в папке app

```
import { Component, Input } from "@angular/core";
import { DiscountService } from "./discount.service";

@Component({
    selector: "paDiscountDisplay",
    template: `<div class="bg-info p-a-1">
                The discount is {{discounter.discount}}
               </div>`
})
export class PaDiscountDisplayComponent {

    @Input("discounter")
    discounter: DiscountService;
}
```

`DiscountDisplayComponent` использует встроенный шаблон для отображения величины скидки, которая берется из объекта `DiscountService`, передаваемого через входное свойство.

Для второго компонента, использующего класс `DiscountService`, добавьте файл discountEditor.component.ts в папку app и включите в него код из листинга 19.4.

Листинг 19.4. Содержимое файла discountEditor.component.ts в папке app

```
import { Component, Input } from "@angular/core";
import { DiscountService } from "./discount.service";

@Component({
```

```
    selector: "paDiscountEditor",
    template: `<div class="form-group">
                    <label>Discount</label>
                    <input [(ngModel)]="discounter.discount"
                         class="form-control" type="number" />
                </div>`
})
export class PaDiscountEditorComponent {

    @Input("discounter")
    discounter: DiscountService;
}
```

Класс `DiscountEditorComponent` использует встроенный шаблон с элементом `input` для редактирования величины скидки. Элемент `input` содержит двустороннюю привязку к свойству `DiscountService.discount`, целью которой является директива `ngModel`. В листинге 19.5 новые компоненты регистрируются в модуле Angular.

Листинг 19.5. Регистрация компонентов в файле app.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { ProductComponent } from "./component";
import { FormsModule, ReactiveFormsModule  } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
import { PaModel } from "./twoway.directive";
import { PaStructureDirective } from "./structure.directive";
import { PaIteratorDirective } from "./iterator.directive";
import { PaCellColor } from "./cellColor.directive";
import { PaCellColorSwitcher } from "./cellColorSwitcher.directive";
import { ProductTableComponent } from "./productTable.component";
import { ProductFormComponent } from "./productForm.component";
import { PaAddTaxPipe } from "./addTax.pipe";
import { PaCategoryFilterPipe } from "./categoryFilter.pipe";
import { PaDiscountDisplayComponent } from "./discountDisplay.component";
import { PaDiscountEditorComponent } from "./discountEditor.component";

@NgModule({
    imports: [BrowserModule, FormsModule, ReactiveFormsModule],
    declarations: [ProductComponent, PaAttrDirective, PaModel,
        PaStructureDirective, PaIteratorDirective,
        PaCellColor, PaCellColorSwitcher, ProductTableComponent,
        ProductFormComponent, PaAddTaxPipe, PaCategoryFilterPipe,
        PaDiscountDisplayComponent, PaDiscountEditorComponent],
    bootstrap: [ProductComponent]
})
export class AppModule { }
```

Чтобы новые компоненты заработали, их необходимо добавить в шаблон родительского компонента. Новый контент нужно разместить под таблицей со списком продуктов; это означает, что файл productTable.component.html необходимо отредактировать, как показано в листинге 19.6.

Листинг 19.6. Добавление элементов компонентов в файле productTable.component.html

```
<table class="table table-sm table-bordered table-striped">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th><th></th></tr>
    <tr *paFor="let item of getProducts(); let i = index;
            let odd = odd; let even = even" [class.bg-info]="odd"
            [class.bg-warning]="even">
        <td style="vertical-align:middle">{{i + 1}}</td>
        <td style="vertical-align:middle">{{item.name}}</td>
        <td style="vertical-align:middle">{{item.category}}</td>
        <td style="vertical-align:middle">
            {{item.price | currency:"USD":true }}
        </td>
        <td class="text-xs-center">
            <button class="btn btn-danger btn-sm" (click)="deleteProduct(item.id)">
                Delete
            </button>
        </td>
    </tr>
</table>

<paDiscountEditor [discounter]="discounter"></paDiscountEditor>
<paDiscountDisplay [discounter]="discounter"></paDiscountDisplay>
```

Эти элементы соответствуют свойствам `selector` компонента в листингах 19.3 и 19.4 и используют привязки данных для задания значений входных свойств. Далее остается создать в родительском компоненте объект, который предоставит значения для выражений привязки данных (листинг 19.7).

Листинг 19.7. Создание совместно используемого объекта в файле productTable.component.ts

```
import { Component, Input, ViewChildren, QueryList } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";
import { DiscountService } from "./discount.service";

@Component({
    selector: "paProductTable",
    templateUrl: "app/productTable.component.html"
})
export class ProductTableComponent {
    discounter: DiscountService = new DiscountService();

    @Input("model")
    dataModel: Model;

    getProduct(key: number): Product {
        return this.dataModel.getProduct(key);
    }

    getProducts(): Product[] {
```

```
        return this.dataModel.getProducts();
    }

    deleteProduct(key: number) {
        this.dataModel.deleteProduct(key);
    }

    dateObject: Date = new Date(2020, 1, 20);
    dateString: string = "2020-02-20T00:00:00.000Z";
    dateNumber: number = 1582156800000;
}
```

На рис. 19.2 изображен контент новых компонентов. Изменения в значении входного элемента, предоставляемого одним компонентом, будут отражаться в привязке со строковой интерполяцией, предоставляемой другим компонентом. В этих изменениях отражается совместное использование объекта `DiscountService` и его свойства `discount`.

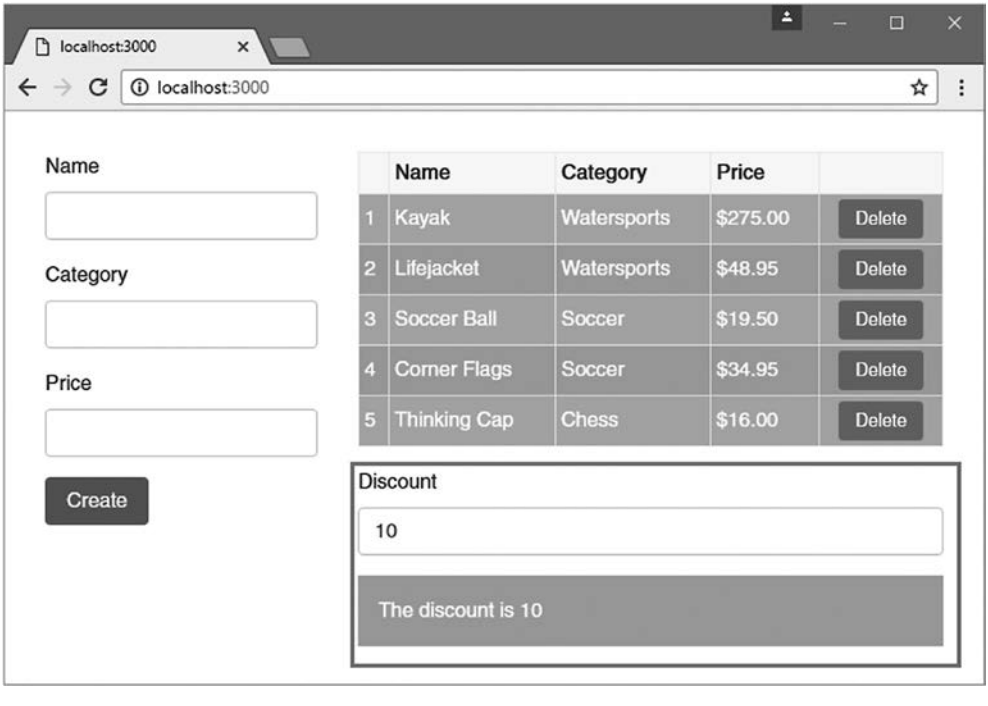


Рис. 19.2. Добавление компонентов в приложение

Процесс добавления новых компонентов и совместно используемого объекта был прямолинейным и логичным — до последней стадии. Проблема связана с механизмом создания и распространения совместно используемого объекта: экземпляра класса `DiscountService`.

Так как Angular изолирует компоненты друг от друга, я не смогу обеспечить совместный доступ к объекту `DiscountService` напрямую из `DiscountEditorComponent` и `DiscountDisplayComponent`. Каждый компонент мог бы создать собственный объект `DiscountService`, но это означает, что изменения в компоненте `DiscountEditorComponent` не будут отображаться в компоненте `DiscountDisplayComponent`.

Это обстоятельство привело меня к решению с созданием объекта `DiscountService` в компоненте таблицы товаров — первом общем предке компонентов редактирования и вывода. Это позволило мне распределить объект `DiscountService` через шаблон компонента таблицы товаров; тем самым гарантируется, что один объект будет совместно использоваться обоими компонентами, которым он нужен.

Но здесь возникает пара проблем. Первая заключается в том, что классу `ProductTableComponent` не нужно иметь или использовать объект `DiscountService` для реализации своей функциональности. Просто так вышло, что он является первым общим предком компонентов, которым нужен этот объект. А создание общего объекта в классе `ProductTableComponent` несколько усложняет этот класс и его эффективное тестирование. Приращение сложности незначительное, но оно сопровождает каждый совместно используемый объект, необходимый приложению, а в сложном приложении количество таких объектов может быть очень большим и каждый из них будет создаваться компонентом, который просто оказался первым общим предком классов, от него зависящих.

На суть второй проблемы намекает термин «первый общий предок». Класс `ProductTableComponent` оказывается родителем обоих классов, зависящих от объекта `DiscountService`; но что произойдет, если я захочу переместить компонент `DiscountEditorComponent`, чтобы он отображался под формой, а не под таблицей? В этой ситуации мне придется перемещаться наверх по дереву компонентов, пока не найдется общий предок, который станет корневым компонентом. А после этого придется перемещаться вниз по дереву компонентов, добавляя входные свойства и изменяя шаблоны так, чтобы каждый промежуточный компонент мог получить объект `DiscountService` от своего родителя и передать его любым дочерним компонентам, потомкам которых он нужен. И это относится ко всем директивам, зависящим от получения объекта `DiscountService`: каждый компонент, шаблон которого содержит привязки данных, целью которых является данная директива, должен проследить за тем, чтобы они тоже были частью цепочки распространения.

В результате возникает тесное сцепление компонентов и директив приложения. Если вам потребуется переместить или повторно использовать компонент в другой части приложения, потребуется серьезный рефакторинг, а задача управления входными свойствами и привязки данных станет неподъемной.

Распределение объектов как служб, использующих внедрение зависимостей

Существует более разумный механизм распределения объектов среди классов, зависящих от них, — *внедрение зависимостей* (dependency injection), при кото-

ром объекты предоставляются классам из внешнего источника. Angular включает встроенную систему внедрения зависимостей и предоставляет внешний источник объектов, называемый *провайдером* (provider). В следующих разделах мы переработаем свой пример, чтобы объект `DiscountService` предоставлялся без необходимости использования иерархии компонентов как механизма распределения.

Подготовка службы

Любой объект, которым вы управляете через механизм внедрения зависимостей, называется *службой* (service); по этой причине я выбрал имя `DiscountService` для класса, определяющего совместно используемый объект, а сам класс определяется в файле с именем discount.service.ts. В Angular классы служб обозначаются декоратором `@Injectable` (листинг 19.8). Декоратор `@Injectable` не определяет никаких свойств конфигурации.

Листинг 19.8. Подготовка класса как службы в файле discount.service.ts

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

@Injectable()
export class DiscountService {
    private discountValue: number = 10;

    public get discount(): number {
        return this.discountValue;
    }

    public set discount(newValue: number) {
        this.discountValue = newValue || 0;
    }

    public applyDiscount(price: number) {
        return Math.max(price - this.discountValue, 5);
    }
}
```

Подготовка зависимых компонентов

Для объявления зависимостей классы используют конструктор. Когда Angular требуется создать экземпляр класса (например, при обнаружении элемента, который соответствует свойству `selector`, определяемому компонентом), анализируется его конструктор и типы всех аргументов. Затем Angular пытается разрешить зависимости, используя имеющиеся определения служб. Термин «внедрение зависимостей» объясняется тем, что каждая зависимость внедряется в конструктор для создания нового экземпляра.

В нашем примере это означает, что компоненты, зависящие от объекта `DiscountService`, уже не нуждаются во входных свойствах и могут объявить зависимость через конструктор. В листинге 19.9 показаны изменения в классе `DiscountDisplayComponent`.

Листинг 19.9. Объявление зависимости в файле discountDisplay.component.ts

```
import { Component, Input } from "@angular/core";
import { DiscountService } from "./discount.service";

@Component({
    selector: "paDiscountDisplay",
    template: `<div class="bg-info p-a-1">
                The discount is {{discounter.discount}}
               </div>`
})
export class PaDiscountDisplayComponent {

    constructor(private discounter: DiscountService) { }
}
```

Те же изменения можно применить к классу `DiscountEditorComponent`: входное свойство заменяется зависимостью, объявляемой через конструктор (листинг 19.10).

Листинг 19.10. Объявление зависимости в файле discountEditor.component.ts

```
import { Component, Input } from "@angular/core";
import { DiscountService } from "./discount.service";

@Component({
    selector: "paDiscountEditor",
    template: `<div class="form-group">
                    <label>Discount</label>
                    <input [(ngModel)]="discounter.discount"
                         class="form-control" type="number" />
               </div>`
})
export class PaDiscountEditorComponent {

    constructor(private discounter: DiscountService) { }
}
```

Изменения незначительны, но они позволяют избежать распределения объектов через шаблоны и входные свойства и способствуют построению более гибкого приложения. Теперь можно удалить объект `DiscountService` из компонента таблицы товаров (листинг 19.11).

Листинг 19.11. Удаление совместно используемого объекта из файла productTable.component.ts

```
import { Component, Input } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";
import { DiscountService } from "./discount.service";

@Component({
    selector: "paProductTable",
    templateUrl: "app/productTable.component.html"
})
```

```
export class ProductTableComponent {
    //discounter: DiscountService = new DiscountService();

    // ...Другие методы и свойства опущены для краткости...
}
```

А поскольку родительский компонент уже не предоставляет совместно используемый объект через привязки данных, можно удалить их из шаблона (листинг 19.12).

Листинг 19.12. Удаление привязки данных из файла productTable.component.html

```
<table class="table table-sm table-bordered table-striped">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th><th></th></tr>
    <tr *paFor="let item of getProducts(); let i = index;
            let odd = odd; let even = even" [class.bg-info]="odd"
            [class.bg-warning]="even">
        <td style="vertical-align:middle">{{i + 1}}</td>
        <td style="vertical-align:middle">{{item.name}}</td>
        <td style="vertical-align:middle">{{item.category}}</td>
        <td style="vertical-align:middle">
            {{item.price | currency:"USD":true }}
        </td>
        <td class="text-xs-center">
            <button class="btn btn-danger btn-sm" (click)="deleteProduct(item.id)">
                Delete
            </button>
        </td>
    </tr>
</table>

<paDiscountEditor></paDiscountEditor>
<paDiscountDisplay></paDiscountDisplay>
```

Регистрация службы

Остается внести последнее изменение — настроить механизм внедрения зависимостей, чтобы он мог предоставлять объекты `DiscountService` компонентам, которым они нужны. Чтобы открыть доступ к службе в приложении, следует зарегистрировать ее в модуле Angular (листинг 19.13).

Листинг 19.13. Регистрация службы в файле app.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { ProductComponent } from "./component";
import { FormsModule, ReactiveFormsModule  } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
import { PaModel } from "./twoway.directive";
import { PaStructureDirective } from "./structure.directive";
import { PaIteratorDirective } from "./iterator.directive";
import { PaCellColor } from "./cellColor.directive";
import { PaCellColorSwitcher } from "./cellColorSwitcher.directive";
```

```
import { ProductTableComponent } from "./productTable.component";
import { ProductFormComponent } from "./productForm.component";
import { PaAddTaxPipe } from "./addTax.pipe";
import { PaCategoryFilterPipe } from "./categoryFilter.pipe";
import { PaDiscountDisplayComponent } from "./discountDisplay.component";
import { PaDiscountEditorComponent } from "./discountEditor.component";
import { DiscountService } from "./discount.service";

@NgModule({
    imports: [BrowserModule, FormsModule, ReactiveFormsModule],
    declarations: [ProductComponent, PaAttrDirective, PaModel,
        PaStructureDirective, PaIteratorDirective,
        PaCellColor, PaCellColorSwitcher, ProductTableComponent,
        ProductFormComponent, PaAddTaxPipe, PaCategoryFilterPipe,
        PaDiscountDisplayComponent, PaDiscountEditorComponent],
    providers: [DiscountService],
    bootstrap: [ProductComponent]
})
export class AppModule { }
```

Свойству `providers` декоратора `NgModule` задается массив классов, которые будут использоваться как службы. На данный момент существует только одна служба, которая предоставляется классом `DiscountService`.

Когда вы сохраните изменения в приложении, внешне ничего не изменится, но механизм внедрения зависимостей будет предоставлять компонентам нужный им объект `DiscountService`.

Анализ изменений при использовании внедрения зависимостей

Angular идеально интегрирует внедрение зависимостей в свою функциональность. Каждый раз, когда Angular встречает элемент, которому нужно создать экземпляр нового структурного блока (например, компонента или канала), по конструктору класса проверяются объявленные зависимости, а его службы используются для разрешения этих зависимостей. Набор служб, используемых для разрешения зависимостей, включает нестандартные службы, определенные приложением (такие, как служба `DiscountService`, зарегистрированная в листинге 19.13), а также набор встроенных служб, предоставляемых Angular; они описаны в последующих главах.

Внесенные изменения — включение внедрения зависимостей из предыдущего раздела — никак не повлияли ни на работу приложения, ни на его внешний вид. Однако существует принципиальное отличие в структуре приложения, с которой оно становится более гибким и подвижным. Лучший способ продемонстрировать этот факт — добавить компоненты, требующие использования `DiscountService`, в другую часть приложения (листинг 19.14).

Листинг 19.14. Добавление компонентов в файл productForm.component.html

```
<form novalidate [formGroup]="form" (ngSubmit)="submitForm(form)">
    <div class="form-group" *ngFor="let control of form.productControls">
        <label>{{control.label}}</label>
```

```
        <input class="form-control"
            [(ngModel)]="newProduct[control.modelProperty]"
            name="{{control.modelProperty}}"
            formControlName="{{control.modelProperty}}" />
        <ul class="text-danger list-unstyled"
                *ngIf="(formSubmitted || control.dirty) && !control.valid">
            <li *ngFor="let error of control.getValidationMessages()">
                {{error}}
            </li>
        </ul>
    </div>
    <button class="btn btn-primary" type="submit"
        [disabled]="formSubmitted && !form.valid"
        [class.btn-secondary]="formSubmitted && !form.valid">
            Create
    </button>
</form>
<paDiscountEditor></paDiscountEditor>
<paDiscountDisplay></paDiscountDisplay>
```

Новые элементы дублируют компоненты `DiscountDisplayComponent` и `DiscountEditorComponent`, чтобы они выводились под формой создания новых продуктов (рис. 19.3).

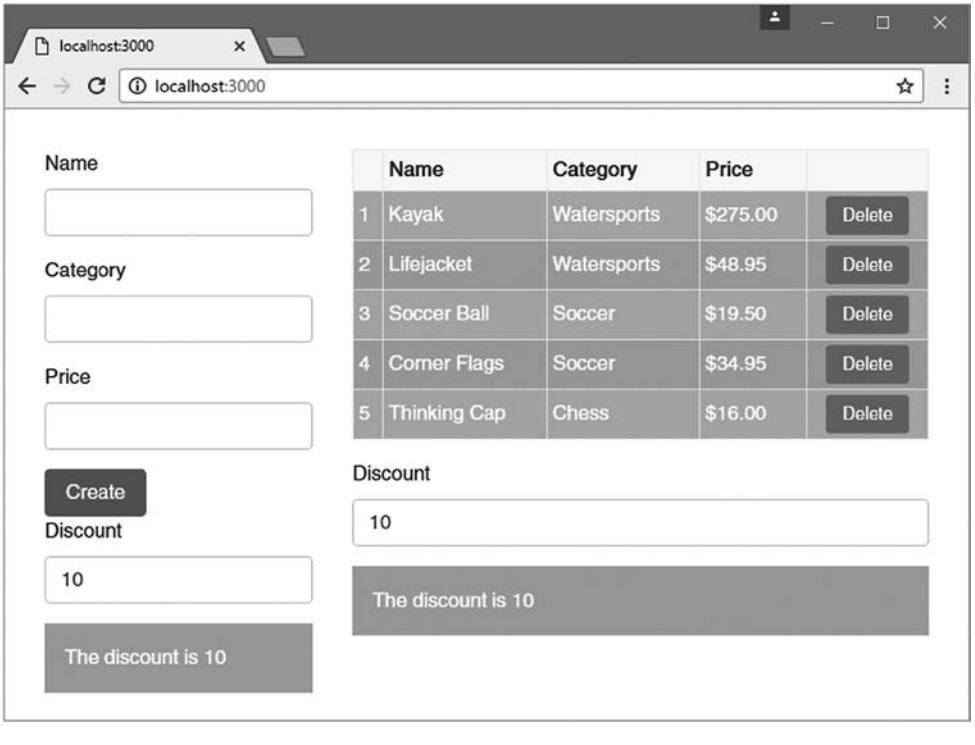


Рис. 19.3. Дублирование компонентов с использованием зависимостей

Здесь заслуживают внимания два момента. Во-первых, благодаря внедрению зависимостей все свелось к простому процессу добавления элементов в шаблон, без необходимости изменения компонентов-предков для предоставления объекта `DiscountService` с использованием входных свойств.

Во-вторых, все компоненты приложения, объявляющие зависимость от `DiscountService`, получают один и тот же объект. Если отредактировать значение в любом из элементов `input`, изменения будут отражены в другом элементе `input` и в привязках со строковой интерполяцией (рис. 19.4).

Рис. 19.4. Доказательство разрешения зависимости на базе совместно используемого объекта

Объявление зависимостей в других структурных блоках

Не только компоненты могут объявлять зависимости через конструктор. После того как вы определили службу, ее можно использовать более широко, в том числе и в других структурных блоках приложения (например, каналах и директивах). Данная возможность будет продемонстрирована ниже.

Объявление зависимости в канале

Каналы могут объявлять зависимости от служб, определяя конструктор с аргументами для всех необходимых служб. Создайте файл с именем discount.pipe.ts в папке app и используйте его для определения канала из листинга 19.15.

Листинг 19.15. Содержимое файла discount.pipe.ts в папке app

```
import { Pipe, Injectable } from "@angular/core";
import { DiscountService } from "./discount.service";

@Pipe({
    name: "discount",
    pure: false
})
export class PaDiscountPipe {

    constructor(private discount: DiscountService) { }

    transform(price: number): number {
        return this.discount.applyDiscount(price);
    }
}
```

Класс `PaDiscountPipe` представляет канал, который получает цену и генерирует результат вызовом метода `DiscountService.applyDiscount`, с передачей службы через конструктор. Свойство `pure` в декораторе `@Pipe` равно `false`; это означает, что канал будет обновлять свой результат при изменении значения, хранимого в `DiscountService`, которое не будет распознаваться процессом обнаружения изменений Angular.

ПРИМЕЧАНИЕ

Как объясняется в главе 18, пользоваться данной возможностью следует осторожно, так как она означает, что метод transform будет вызываться после каждого изменения в приложении — не только при изменении службы.

В листинге 19.16 показано, как новый канал регистрируется в модуле Angular приложения.

Листинг 19.16. Регистрация канала в файле app.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { ProductComponent } from "./component";
import { FormsModule, ReactiveFormsModule  } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
import { PaModel } from "./twoway.directive";
import { PaStructureDirective } from "./structure.directive";
import { PaIteratorDirective } from "./iterator.directive";
import { PaCellColor } from "./cellColor.directive";
import { PaCellColorSwitcher } from "./cellColorSwitcher.directive";
import { ProductTableComponent } from "./productTable.component";
import { ProductFormComponent } from "./productForm.component";
import { PaAddTaxPipe } from "./addTax.pipe";
import { PaCategoryFilterPipe } from "./categoryFilter.pipe";
import { PaDiscountDisplayComponent } from "./discountDisplay.component";
import { PaDiscountEditorComponent } from "./discountEditor.component";
import { DiscountService } from "./discount.service";
import { PaDiscountPipe } from "./discount.pipe";

@NgModule({
    imports: [BrowserModule, FormsModule, ReactiveFormsModule],
    declarations: [ProductComponent, PaAttrDirective, PaModel,
        PaStructureDirective, PaIteratorDirective,
        PaCellColor, PaCellColorSwitcher, ProductTableComponent,
        ProductFormComponent, PaAddTaxPipe, PaCategoryFilterPipe,
        PaDiscountDisplayComponent, PaDiscountEditorComponent,
        PaDiscountPipe],
    providers: [DiscountService],
    bootstrap: [ProductComponent]
})
export class AppModule { }
```

В листинге 19.17 новый канал применяется к столбцу цены таблицы товаров.

Листинг 19.17. Применение канала в файле productTable.component.html

```
<table class="table table-sm table-bordered table-striped">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th><th></th></tr>
    <tr *paFor="let item of getProducts(); let i = index;
            let odd = odd; let even = even" [class.bg-info]="odd"
            [class.bg-warning]="even">
        <td style="vertical-align:middle">{{i + 1}}</td>
        <td style="vertical-align:middle">{{item.name}}</td>
        <td style="vertical-align:middle">{{item.category}}</td>
        <td style="vertical-align:middle">
            {{item.price | discount | currency:"USD":true }}
        </td>
        <td class="text-xs-center">
            <button class="btn btn-danger btn-sm" (click)="deleteProduct(item.id)">
                Delete
            </button>
        </td>
    </tr>
</table>

<paDiscountEditor></paDiscountEditor>
<paDiscountDisplay></paDiscountDisplay>
```

Канал `discount` применяет к цене скидку, а затем передает значение каналу `currency` для форматирования. Чтобы понаблюдать за эффектом использования службы в канале, измените значение в одном из элементов `input` (рис. 19.5).

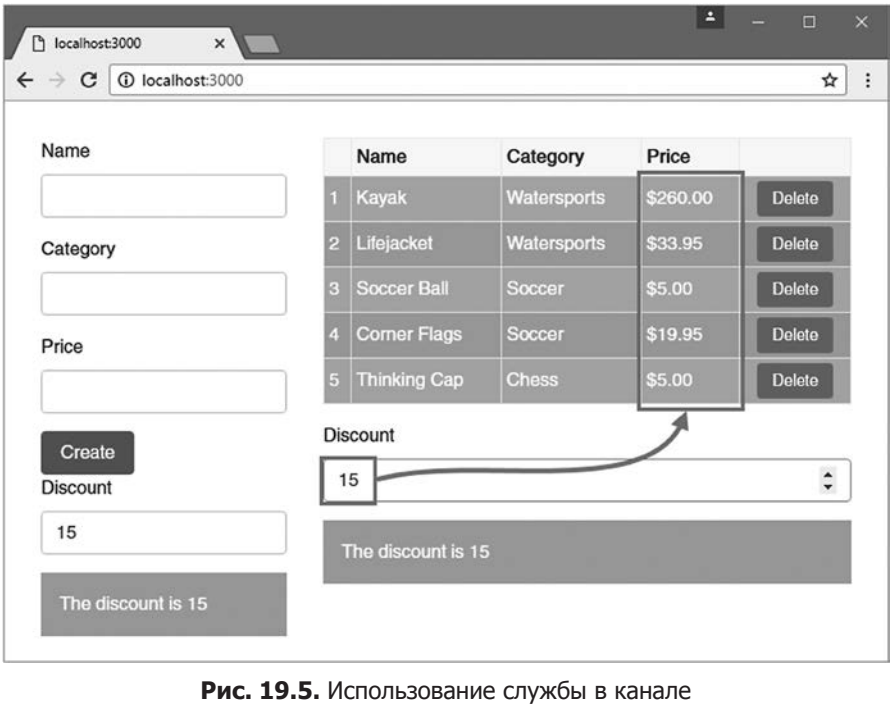


Рис. 19.5. Использование службы в канале

Объявление зависимостей в директивах

Директивы тоже могут использовать службы. Как я объяснял в главе 17, компоненты — всего лишь директивы с шаблонами, поэтому все, что работает с компонентом, также будет работать и с директивой.

Чтобы продемонстрировать использование службы в директиве, создайте файл discountAmount.directive.ts в папке app и включите в него определение директивы из листинга 19.18.

Листинг 19.18. Содержимое файла discountAmount.directive.ts в папке app

```
import { Directive, HostBinding, Input,
    SimpleChange, KeyValueDiffer, KeyValueDiffers,
    ChangeDetectorRef } from "@angular/core";
import { DiscountService } from "./discount.service";

@Directive({
    selector: "td[pa-price]",
    exportAs: "discount"
})
export class PaDiscountAmountDirective {
    private differ: KeyValueDiffer;

    constructor(private keyValueDiffers: KeyValueDiffers,
        private changeDetector: ChangeDetectorRef,
        private discount: DiscountService) { }

    @Input("pa-price")
    originalPrice: number;

    discountAmount: number;

    ngOnInit() {
        this.differ =
            this.keyValueDiffers.find(this.discount).create(this.changeDetector);
    }

    ngOnChanges(changes: { [property: string]: SimpleChange }) {
        if (changes["originalPrice"] != null) {
            this.updateValue();
        }
    }

    ngDoCheck() {
        if (this.differ.diff(this.discount) != null) {
            this.updateValue();
        }
    }

    private updateValue() {
        this.discountAmount = this.originalPrice
            - this.discount.applyDiscount(this.originalPrice);
    }
}
```

У директив не существует аналога свойства pure, используемого каналами, поэтому они несут полную ответственность за реакцию на изменения, распространяемые через службы. Некоторые службы, такие как встроенная служба Http, описанная в главе 24, поддерживают механизм уведомлений о происходящих изменениях. Другие, такие как служба DiscountService, использованная в примере, оставляют всю работу директиве.

Директива выводит цену товара со скидкой. Свойство selector находит элементы td с атрибутом pa-price, который также используется в качестве входного свойства для получения цены. Директива экспортирует свою функциональность через свойство exportAs и предоставляет свойство discountAmount, которому присваивается величина скидки для товара.

В этой директиве заслуживают внимания еще два момента. Во-первых, объект DiscountService не единственный параметр конструктора в классе директивы.

```
...
constructor(private keyValueDiffers: KeyValueDiffers,
            private changeDetector: ChangeDetectorRef,
            private discount: DiscountService) { }
...
```

Параметры KeyValueDiffers и ChangeDetectorRef также являются зависимостями, которые Angular попытается разрешить при создании нового элемента класса директивы. Это примеры встроенных служб, которые Angular предоставляет для функциональности стандартных задач.

Второй момент, заслуживающий внимания, — то, что делает директива с получаемыми службами. Компонентам и каналу, использующим службу DiscountService, не нужно беспокоиться об отслеживании изменений — либо потому, что Angular автоматически вычисляет выражения привязок данных и обновляет их при изменении скидки (для компонентов), либо потому, что любое изменение в приложении инициирует обновление (для нечистого канала).

Привязка данных этой директивы работает через свойство price, модификация которого инициирует изменение. Однако также существует зависимость от свойства discount, определяемого классом DiscountService. Изменения в свойстве discount обнаруживаются с использованием служб, получаемых через конструктор; они похожи на те, которые использовались для отслеживания изменений в итеративных последовательностях (см. главу 16), но работают с парами объектов «ключ — значение» (как объект Map) или обычными объектами, определяющими свойства (как DiscountService). Когда Angular вызывает метод ngDoCheck, директива использует диффер пары «ключ — значение», чтобы узнать о наличии изменений. (Также можно было воспользоваться отслеживанием текущего обновления в классе директивы, но я хотел представить пример использования диффера «ключ — значение».) Директива также реализует метод ngOnChanges, чтобы реагировать на изменения в значении входного свойства. Для обновлений обоих видов вызывается метод updateValue, который вычисляет цену со скидкой и задает ее свойству discountAmount. В листинге 19.19 новая директива регистрируется в модуле Angular приложения.

Листинг 19.19. Регистрация директивы в файле app.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { ProductComponent } from "./component";
import { FormsModule, ReactiveFormsModule  } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
import { PaModel } from "./twoway.directive";
import { PaStructureDirective } from "./structure.directive";
import { PaIteratorDirective } from "./iterator.directive";
import { PaCellColor } from "./cellColor.directive";
import { PaCellColorSwitcher } from "./cellColorSwitcher.directive";
import { ProductTableComponent } from "./productTable.component";
import { ProductFormComponent } from "./productForm.component";
import { PaAddTaxPipe } from "./addTax.pipe";
import { PaCategoryFilterPipe } from "./categoryFilter.pipe";
import { PaDiscountDisplayComponent } from "./discountDisplay.component";
import { PaDiscountEditorComponent } from "./discountEditor.component";
import { DiscountService } from "./discount.service";
import { PaDiscountPipe } from "./discount.pipe";
import { PaDiscountAmountDirective } from "./discountAmount.directive";

@NgModule({
    imports: [BrowserModule, FormsModule, ReactiveFormsModule],
    declarations: [ProductComponent, PaAttrDirective, PaModel,
        PaStructureDirective, PaIteratorDirective,
        PaCellColor, PaCellColorSwitcher, ProductTableComponent,
        ProductFormComponent, PaAddTaxPipe, PaCategoryFilterPipe,
        PaDiscountDisplayComponent, PaDiscountEditorComponent,
        PaDiscountPipe, PaDiscountAmountDirective],
    providers: [DiscountService],
    bootstrap: [ProductComponent]
})
export class AppModule { }
```

В листинге 19.20 в таблицу добавляется новый столбец, привязка со строковой интерполяцией используется для обращения к свойству, предоставляемому директивой, а результат передается каналу `currency`.

Листинг 19.20. Создание нового столбца в файле productTable.component.html

```
<table class="table table-sm table-bordered table-striped">
    <tr>
        <th></th><th>Name</th><th>Category</th><th>Price</th>
        <th>Discount</th><th></th>
    </tr>
    <tr *paFor="let item of getProducts(); let i = index;
            let odd = odd; let even = even" [class.bg-info]="odd"
            [class.bg-warning]="even">
        <td style="vertical-align:middle">{{i + 1}}</td>
        <td style="vertical-align:middle">{{item.name}}</td>
        <td style="vertical-align:middle">{{item.category}}</td>
        <td style="vertical-align:middle">
            {{item.price | discount | currency:"USD":true }}
        </td>
```

```
        <td style="vertical-align:middle" [pa-price]="item.price"
                #discount="discount">
            {{ discount.discountAmount | currency:"USD": true }}
        </td>
        <td class="text-xs-center">
            <button class="btn btn-danger btn-sm" (click)="deleteProduct(item.id)">
                Delete
            </button>
        </td>
    </tr>
</table>

<paDiscountEditor></paDiscountEditor>
<paDiscountDisplay></paDiscountDisplay>
```

Директива также могла создать привязку к свойству `textContent` для задания контента своего управляющего элемента, но это не позволило бы использовать канал `currency`. Вместо этого директива присваивается переменной шаблона `discount`, которая затем используется в привязке со строковой интерполяцией для обращения к значению `discountAmount` и его последующего форматирования. Результаты показаны на рис. 19.6. Изменения в величине скидки в любом из элементов `input` будут отражены в новом столбце таблицы.

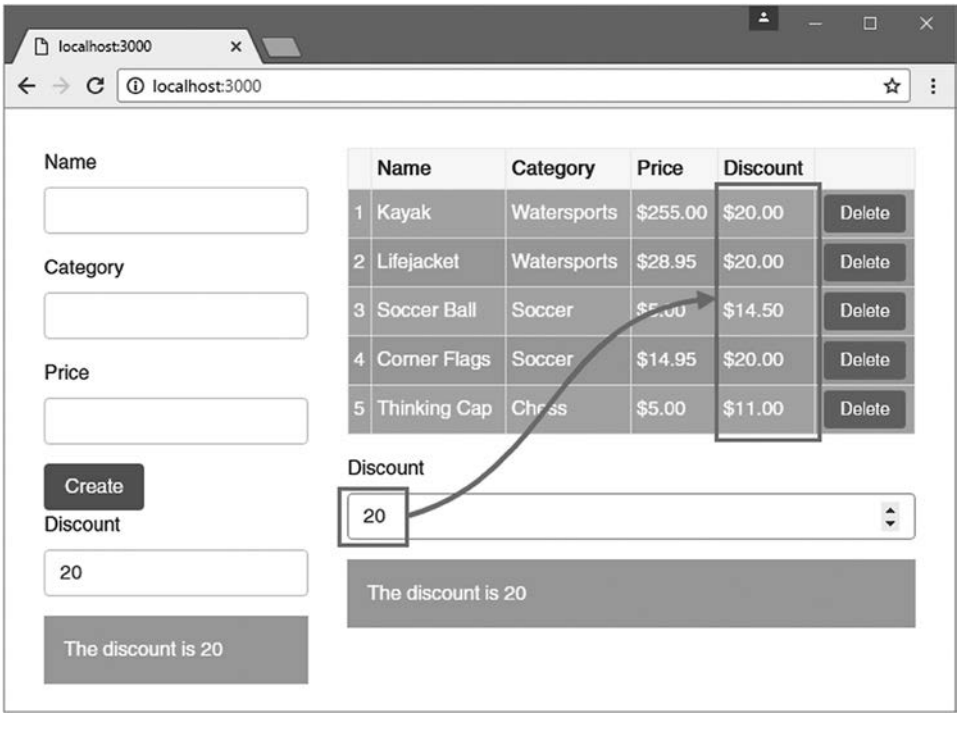


Рис. 19.6. Использование службы в директиве

Проблема изоляции тестов

В приложении также встречается родственная проблема, для решения которой могут использоваться службы и внедрение зависимостей. Вспомните, как класс `Model` создается в корневом компоненте:

```
import { ApplicationRef, Component } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";
import { ProductFormGroup } from "./form.model";

@Component({
    selector: "app",
    templateUrl: "app/template.html"
})

export class ProductComponent {
    model: Model = new Model();

    addProduct(p: Product) {
        this.model.saveProduct(p);
    }
}
```

Корневой компонент определяется как класс `ProductComponent`, а значение его свойства `model` задается созданием нового экземпляра класса `Model`. Такое решение работает (и это абсолютно нормальный способ создания объектов!), но оно усложняет эффективное выполнение модульных тестов.

Модульное тестирование лучше всего работает тогда, когда вы можете изолировать одну небольшую часть приложения от остального кода и сосредоточиться на ней в процессе тестирования. Но когда вы создаете экземпляр класса `ProductComponent`, вы также неявно создаете экземпляр класса `Model`. Если в ходе тестирования метода `addProduct` обнаружится проблема, вы не будете знать, в каком классе она кроется — `ProductComponent` или `Model`.

Изоляция компонентов с использованием служб и внедрение зависимостей

Основная проблема заключается в том, что класс `ProductComponent` тесно связывается с классом `Model`, который, в свою очередь, тесно связывается с классом `SimpleDataSource`. Внедрение зависимостей может использоваться для разрыва связей между структурными блоками приложения, чтобы каждый класс можно было изолировать и протестировать отдельно от других. В последующих разделах рассматривается процесс разделения этих тесно связанных классов; он проходит практически по той же схеме, что и в предыдущем разделе, но с более глубоким анализом приложения.

Подготовка служб

Декоратор `@Injectable` используется для пометки служб, как и в предыдущем примере. В листинге 19.21 показано применение декоратора к классу `SimpleDataSource`.

Листинг 19.21. Пометка службы в файле datasource.model.ts

```
import { Injectable } from "@angular/core";
import { Product } from "./product.model";

@Injectable()
export class SimpleDataSource {
    private data:Product[];

    constructor() {
        this.data = new Array<Product>(
        new Product(1, "Kayak", "Watersports", 275),
        new Product(2, "Lifejacket", "Watersports", 48.95),
        new Product(3, "Soccer Ball", "Soccer", 19.50),
        new Product(4, "Corner Flags", "Soccer", 34.95),
        new Product(5, "Thinking Cap", "Chess", 16));
    }

    getData(): Product[] {
        return this.data;
    }
}
```

Никакие другие изменения не потребуются. В листинге 19.22 тот же декоратор применяется к репозиторию данных; так как этот класс зависит от класса `SimpleDataSource`, он объявляется как зависимость, передаваемая через конструктор, вместо прямого создания экземпляра.

Листинг 19.22. Пометка службы и объявление зависимости в файле repository.model.ts

```
import { Injectable } from "@angular/core";
import { Product } from "./product.model";
import { SimpleDataSource } from "./datasource.model";

@Injectable()
export class Model {
    //private dataSource: SimpleDataSource;
    private products: Product[];
    private locator = (p:Product, id:number) => p.id == id;

    constructor(private dataSource: SimpleDataSource) {
        //this.dataSource = new SimpleDataSource();
        this.products = new Array<Product>();
        this.dataSource.getData().forEach(p => this.products.push(p));
    }

    // ...Другие свойства и методы опущены для краткости...
}
```

Обратите внимание на один важный момент: службы могут объявлять зависимости от других служб. Когда возникает необходимость в создании нового экземпляра класса службы, Angular анализирует конструктор и пытается разрешить службы по тем же правилам, как для компонентов или директив.

Регистрация служб

Чтобы среда Angular знала, как разрешать зависимости от служб, их необходимо зарегистрировать (листинг 19.23).

Листинг 19.23. Регистрация служб в файле app.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { ProductComponent } from "./component";
import { FormsModule, ReactiveFormsModule  } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
import { PaModel } from "./twoway.directive";
import { PaStructureDirective } from "./structure.directive";
import { PaIteratorDirective } from "./iterator.directive";
import { PaCellColor } from "./cellColor.directive";
import { PaCellColorSwitcher } from "./cellColorSwitcher.directive";
import { ProductTableComponent } from "./productTable.component";
import { ProductFormComponent } from "./productForm.component";
import { PaAddTaxPipe } from "./addTax.pipe";
import { PaCategoryFilterPipe } from "./categoryFilter.pipe";
import { PaDiscountDisplayComponent } from "./discountDisplay.component";
import { PaDiscountEditorComponent } from "./discountEditor.component";
import { DiscountService } from "./discount.service";
import { PaDiscountPipe } from "./discount.pipe";
import { PaDiscountAmountDirective } from "./discountAmount.directive";
import { SimpleDataSource } from "./datasource.model";
import { Model } from "./repository.model";

@NgModule({
    imports: [BrowserModule, FormsModule, ReactiveFormsModule],
    declarations: [ProductComponent, PaAttrDirective, PaModel,
        PaStructureDirective, PaIteratorDirective,
        PaCellColor, PaCellColorSwitcher, ProductTableComponent,
        ProductFormComponent, PaAddTaxPipe, PaCategoryFilterPipe,
        PaDiscountDisplayComponent, PaDiscountEditorComponent,
        PaDiscountPipe, PaDiscountAmountDirective],
    providers: [DiscountService, SimpleDataSource, Model],
    bootstrap: [ProductComponent]
})
export class AppModule { }
```

Подготовка зависимого компонента

Вместо того чтобы создавать объект `Model` напрямую, корневой компонент может объявить зависимость через конструктор. Angular разрешает такую зависимость

с использованием механизма внедрения зависимостей при запуске приложения (листинг 19.24).

Листинг 19.24. Объявление зависимости от службы в файле component.ts

```
import { ApplicationRef, Component } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";
import { ProductFormGroup } from "./form.model";

@Component({
    selector: "app",
    templateUrl: "app/template.html"
})
export class ProductComponent {
    //model: Model = new Model();

    constructor(private model: Model) { }

    addProduct(p: Product) {
        this.model.saveProduct(p);
    }
}
```

Образуется цепочка зависимостей, которую Angular необходимо разрешить. При запуске приложения модуль Angular указывает, что классу `ProductComponent` понадобится объект `Model`. Angular проверяет класс `Model` и обнаруживает, что ему нужен объект `SimpleDataSource`. Angular проверяет объект `SimpleDataSource` и обнаруживает, что у него нет объявленных зависимостей, а следовательно, был достигнут конец цепочки. Angular создает объект `SimpleDataSource` и передает его в аргументе конструктора `Model` для создания объекта `Model`, который затем может быть передан конструктору класса `ProductComponent` для создания объекта, который будет использоваться в качестве корневого компонента. Все это происходит автоматически на основании конструкторов, определяемых каждым классом, и использовании декоратора `@Injectable`.

Эти изменения не отражаются на внешнем виде приложения, но позволяют совершенно по-новому подойти к проведению модульных тестов. Класс `ProductComponent` требует, чтобы объект `Model` передавался в аргументе конструктора, что позволяет использовать фиктивный (mock) объект.

Разбиение прямых зависимостей между классами в приложении означает, что каждый класс можно изолировать для проведения модульного тестирования с передачей фиктивных аргументов конструктору. Это открывает возможность логически согласованной, стабильной проверки функциональности методов или других аспектов.

Переход на работу со службами

Когда вы используете службы в своем приложении, процесс обычно начинает жить собственной жизнью. Вскоре вы начнете анализировать отношения между

структурными блоками, которые вы создали. Степень применения служб (по крайней мере частично) является делом личного вкуса.

Хорошим примером служит использование класса `Model` в корневом компоненте. Хотя компонент реализует метод, использующий объект `Model`, он делает это потому, что он должен обрабатывать нестандартное событие от одного из своих дочерних компонентов. Другая (и последняя) причина, по которой корневому компоненту нужен объект `Model`, заключается в том, что он должен передать его через шаблон другому дочернему компоненту через входное свойство.

Это нельзя назвать огромной проблемой. Возможно, вы решите допустить такие отношения в своем проекте. В конце концов, каждый из компонентов можно изолировать для модульного тестирования, и отношения между ними могут передавать некоторый смысл (хотя и ограниченный). Отношения такого рода между компонентами помогут прояснить функциональность, предоставляемую приложением.

С другой стороны, чем больше вы используете службы, тем скорее структурные блоки вашего проекта превратятся в автономные блоки функциональности, пригодные для многократного использования, что может упростить процесс добавления или изменения функциональности по мере становления проекта.

Однозначно правильных или неправильных решений не бывает. Вы должны найти тот баланс, который подходит лично вам, вашей команде, а в конечном итоге пользователям и заказчикам. Не всем нравится внедрение зависимостей, и не все проводят модульное тестирование.

Я предпочитаю как можно шире применять внедрение зависимостей. Я выяснил, что итоговая структура приложений может значительно отличаться от той, что я ожидаю при создании нового проекта, а гибкость, обеспечиваемая внедрением зависимостей, помогает избегать многократного рефакторинга. В завершение этой главы мы распространим использование службы `Model` в остальном коде приложения, разрушая связи между корневым компонентом и его непосредственными потомками.

Обновление корневого компонента и шаблона

Изменения начинаются с удаления объекта `Model` из корневого компонента — вместе с методом, в котором он используется, и входным свойством шаблона, которое распространяет модель в один из дочерних компонентов. В листинге 19.25 показаны изменения в классе компонента.

Листинг 19.25. Удаление объекта Model из файла component.ts

```
import { Component } from "@angular/core";
//import { Model } from "./repository.model";
//import { Product } from "./product.model";
//import { ProductFormGroup } from "./form.model";

@Component({
    selector: "app",
```

```
    templateUrl: "app/template.html"
})
export class ProductComponent {
    //model: Model = new Model();

    //constructor(private model: Model) { }

    //addProduct(p: Product) {
    //    this.model.saveProduct(p);
    //}
}
```

Переработанный класс корневого компонента не определяет никакую функциональность. Он существует только для того, чтобы предоставить высокоуровневый контент приложения в своем шаблоне. В листинге 19.26 показаны соответствующие изменения корневого шаблона с удалением привязки нестандартного события и входного свойства.

Листинг 19.26. Удаление привязки данных в файле template.html

```
<div class="col-xs-4 p-a-1">
    <paProductForm></paProductForm>
</div>
<div class="col-xs-8 p-a-1">
    <paProductTable></paProductTable>
</div>
```

Обновление дочерних компонентов

Компонент, который предоставляет форму для создания новых объектов `Product`, зависит от корневого компонента, который обрабатывает нестандартное событие и обновляет модель. Без этой поддержки компонент теперь должен объявить зависимость от `Model` и выполнить обновление самостоятельно (листинг 19.27).

Листинг 19.27. Прямые операции с Model в файле productForm.component.ts

```
import { Component, Output, EventEmitter, ViewEncapsulation } from "@angular/core";
import { Product } from "./product.model";
import { ProductFormGroup } from "./form.model";
import { Model } from "./repository.model";

@Component({
    selector: "paProductForm",
    templateUrl: "app/productForm.component.html",
    //styleUrls: ["app/productForm.component.css"],
    //encapsulation: ViewEncapsulation.Emulated
})
export class ProductFormComponent {
    form: ProductFormGroup = new ProductFormGroup();
    newProduct: Product = new Product();
    formSubmitted: boolean = false;

    constructor(private model: Model) { }
```

```
    //@Output("paNewProduct")
    //newProductEvent = new EventEmitter<Product>();

    submitForm(form: any) {
        this.formSubmitted = true;
        if (form.valid) {
            //this.newProductEvent.emit(this.newProduct);
            this.model.saveProduct(this.newProduct);
            this.newProduct = new Product();
            this.form.reset();
            this.formSubmitted = false;
        }
    }
}
```

Компонент, управляющий таблицей товаров, ранее использовал входное свойство для получения объекта `Model` от родителя; теперь он получает его напрямую, объявляя зависимость через конструктор (листинг 19.28).

Листинг 19.28. Объявление зависимости от Model в файле productTable.component.ts

```
import { Component, Input } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";
import { DiscountService } from "./discount.service";

@Component({
    selector: "paProductTable",
    templateUrl: "app/productTable.component.html"
})
export class ProductTableComponent {
    //discounter: DiscountService = new DiscountService();

    constructor(private dataModel: Model) { }

    //@Input("model")
    //dataModel: Model;

    getProduct(key: number): Product {
        return this.dataModel.getProduct(key);
    }

    getProducts(): Product[] {
        return this.dataModel.getProducts();
    }

    deleteProduct(key: number) {
        this.dataModel.deleteProduct(key);
    }

    dateObject: Date = new Date(2020, 1, 20);
    dateString: string = "2020-02-20T00:00:00.000Z";
    dateNumber: number = 1582156800000;
}
```

После того как все изменения будут сохранены, а браузер перезагрузит приложение Angular, в окне будет отображаться та же функциональность, но логическая структура этой функциональности заметно изменилась: каждый компонент получает нужные ему совместные объекты через механизм внедрения зависимостей, не полагаясь на то, что родительский компонент предоставит их.

Итоги

В этой главе я объяснил проблемы, для решения которых используется механизм внедрения зависимостей, и продемонстрировал процесс определения и использования служб. Вы узнали, как использовать службы для повышения гибкости структуры приложения и как внедрение зависимостей способствует изоляции структурных элементов для эффективного проведения модульного тестирования. В следующей главе будут описаны расширенные возможности работы со службами, предоставляемые Angular.

20 Провайдеры служб

В главе 19 я рассказал о службах и объяснил, как организуется доступ к ним через механизм внедрения зависимостей. При использовании внедрения зависимостей объекты, используемые для разрешения зависимостей, создаются *провайдерами служб*, которые обычно называются просто *провайдерами*. В этой главе вы узнаете, как работают провайдеры, какие типы провайдеров существуют и как создавать провайдеры в различных частях приложения для изменения поведения служб. В табл. 20.1 провайдеры служб представлены в контексте.

Таблица 20.1. Провайдеры служб в контексте

Вопрос	Ответ
Что это такое?	Провайдеры — классы, создающие объекты служб в тот момент, когда Angular впервые потребуется разрешить зависимость
Для чего они нужны?	Провайдеры позволяют создавать объекты служб, адаптированные к потребностям приложения. Простейший провайдер просто создает экземпляр заданного класса, но существуют другие провайдеры с возможностью настройки процесса создания и определения конфигурации объектов служб
Как они используются?	Провайдеры определяются в свойстве providers декоратора модуля Angular. Они также могут определяться компонентами и директивами для предоставления служб их потомкам (см. раздел «Использование локальных провайдеров»)
Есть ли у них недостатки или скрытые проблемы?	Поведение провайдеров может оказаться неожиданным, особенно при работе с локальными провайдерами. Столкнувшись с проблемами, проверьте область видимости созданных вами локальных провайдеров — и зависимости и провайдеры используют одинаковые маркеры
Есть ли альтернативы?	Многим приложениям достаточно базовой функциональности внедрения зависимостей (см. главу 19). Используйте функциональность, описанную в этой главе, только в том случае, если приложение не может быть построено на основе лишь базовой функциональности и только если вы хорошо понимаете, как работает внедрение зависимостей

НЕ ПРОПУСТИТЬ ЛИ ЭТУ ГЛАВУ?

Внедрение зависимостей вызывает сильные эмоции у разработчиков, а мнения высказываются прямо противоположные. Если вы недавно познакомились с внедрением зависимостей и еще не успели сформировать собственное мнение, возможно, эту главу стоит пропустить и ограничиться функциональностью из главы 19. Дело в том, что именно из-за возможностей, описанных в этой главе, многие разработчики опасаются внедрения зависимостей и стараются обходиться без него.

Базовые возможности внедрения зависимостей в Angular понятны и у них есть очевидное преимущество: простота написания и сопровождения кода приложений. Возможности, описанные в этой главе, позволяют точно управлять процессом внедрения зависимостей, но они также резко увеличивают сложность приложения, а в конечном итоге отменяют многие преимущества базовой функциональности.

Если вы решите, что хотите знать все технические подробности внедрения зависимостей, — продолжайте читать. Но если вы недавно пришли в мир внедрения зависимостей, возможно, вам стоит пропустить эту главу, пока вы не придете к выводу, что базовая функциональность из главы 19 не позволяет добиться желаемого результата.

В табл. 20.2 приведена краткая сводка материала главы.

Таблица 20.2. Сводка материала главы

Проблема	Решение	Листинг
Изменение способа создания служб	Используйте провайдер службы	1–3
Определение службы с использованием класса	Используйте провайдер класса	4–6, 10–13
Определение произвольных маркеров для служб	Используйте класс OpaqueToken	7–9
Определение службы с использованием объекта	Используйте провайдер значения	14–15
Определение службы с использованием функции	Используйте провайдер фабрики	16–18
Определение одной службы с использованием другой	Используйте провайдер существующей службы	19
Изменение области видимости службы	Используйте локальный провайдер службы	20–28
Управление разрешением зависимостей	Используйте декоратор @Host, @Optional или @SkipSelf	29–30

Подготовка проекта

Как и в других главах этой части книги, мы продолжим работать с проектом, который был создан в главе 11 и в последний раз был изменен в главе 19. Чтобы подготовить проект, создайте файл с именем log.service.ts в папке app и включите в него определение службы из листинга 20.1.

Листинг 20.1. Содержимое файла log.service.ts в папке app

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

export enum LogLevel {
    DEBUG, INFO, ERROR
}

@Injectable()
export class LogService {
    minimumLevel: LogLevel = LogLevel.INFO;

    logInfoMessage(message: string) {
        this.logMessage(LogLevel.INFO, message);
    }

    logDebugMessage(message: string) {
        this.logMessage(LogLevel.DEBUG, message);
    }

    logErrorMessage(message: string) {
        this.logMessage(LogLevel.ERROR, message);
    }

    logMessage(level: LogLevel, message: string) {
        if (level >= this.minimumLevel) {
            console.log(`Message (${LogLevel[level]}): ${message}`);
        }
    }
}
```

Служба выводит на консоль JavaScript браузера сообщения разной степени серьезности. Мы зарегистрируем и будем использовать эту службу позднее в этой главе.

Когда вы создадите службу и сохраните изменения, выполните следующую команду из папки example, чтобы запустить компилятор TypeScript и сервер HTTP для разработки:

```
npm start
```

На экране появляется новое окно браузера с приложением (рис. 20.1).

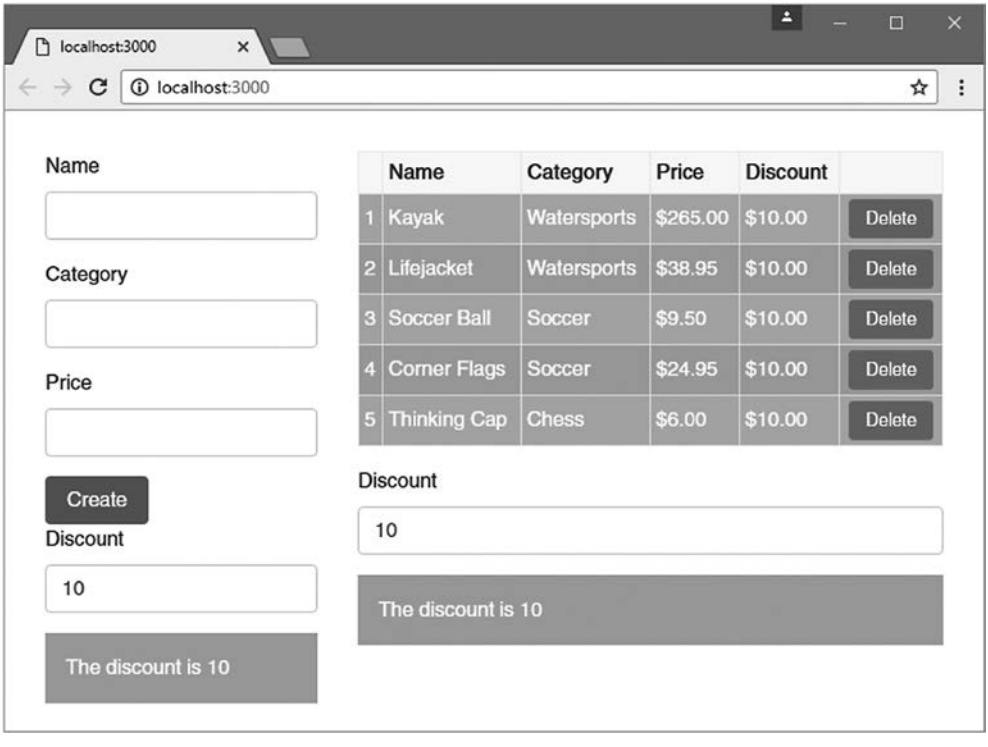


Рис. 20.1. Запуск приложения

Использование провайдеров служб

Как объяснялось в предыдущих главах, классы объявляют зависимости от служб при помощи аргументов конструкторов. Когда потребуется создать новый экземпляр класса, Angular анализирует конструктор и использует комбинацию встроенных и нестандартных служб для разрешения каждого аргумента. В листинге 20.2 представлена обновленная версия класса `DiscountService`, которая зависит от класса `LogService`, созданного в предыдущем разделе.

Листинг 20.2. Создание зависимости в файле discount.service.ts

```
import { Injectable } from "@angular/core";
import { LogService } from "./log.service";

@Injectable()
export class DiscountService {
    private discountValue: number = 10;

    constructor(private logger: LogService) { }

    public get discount(): number {
```

```
        return this.discountValue;
    }

    public set discount(newValue: number) {
        this.discountValue = newValue || 0;
    }

    public applyDiscount(price: number) {
        this.logger.logInfoMessage(`Discount ${this.discount}`
            + ` applied to price: ${price}`);
        return Math.max(price - this.discountValue, 5);
    }
}
```

С изменениями в листинге 20.2 приложение не запускается. Angular обрабатывает документ HTML и начинает строить иерархию компонентов. Каждый компонент имеет собственный шаблон с директивами и привязками данных. Со временем Angular обнаруживает классы, зависящие от класса `DiscountService`. Но создать экземпляр `DiscountService` не удается, потому что конструктору этого класса нужен объект `LogService`, а Angular не знает, что делать с этим классом.

При сохранении изменений из листинга 20.2 на консоль JavaScript в браузере выводится сообщение об ошибке, которое выглядит примерно так:

```
Error in app/productForm.component.html:20:0 caused by: No provider for LogService!
```

Angular делегирует ответственность за создание объектов, необходимых для внедрения зависимостей, *провайдерам*, каждый из которых управляет одним типом зависимостей. Когда потребуется создать экземпляр класса `DiscountService`, Angular ищет подходящего провайдера для разрешения зависимости `LogService`. Такого провайдера нет; Angular не может создать объекты, необходимые для запуска приложения, и выводит сообщение об ошибке.

Самый простой способ создания провайдера — включение класса службы в массив, назначаемый свойству `providers` модуля Angular (листинг 20.3).

Листинг 20.3. Создание провайдера в файле app.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { ProductComponent } from "./component";
import { FormsModule, ReactiveFormsModule  } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
import { PaModel } from "./twoway.directive";
import { PaStructureDirective } from "./structure.directive";
import { PaIteratorDirective } from "./iterator.directive";
import { PaCellColor } from "./cellColor.directive";
import { PaCellColorSwitcher } from "./cellColorSwitcher.directive";
import { ProductTableComponent } from "./productTable.component";
import { ProductFormComponent } from "./productForm.component";
import { PaAddTaxPipe } from "./addTax.pipe";
import { PaCategoryFilterPipe } from "./categoryFilter.pipe";
import { PaDiscountDisplayComponent } from "./discountDisplay.component";
```

```
import { PaDiscountEditorComponent } from "./discountEditor.component";
import { DiscountService } from "./discount.service";
import { PaDiscountPipe } from "./discount.pipe";
import { PaDiscountAmountDirective } from "./discountAmount.directive";
import { SimpleDataSource } from "./datasource.model";
import { Model } from "./repository.model";
import { LogService } from "./log.service";

@NgModule({
    imports: [BrowserModule, FormsModule, ReactiveFormsModule],
    declarations: [ProductComponent, PaAttrDirective, PaModel,
        PaStructureDirective, PaIteratorDirective,
        PaCellColor, PaCellColorSwitcher, ProductTableComponent,
        ProductFormComponent, PaAddTaxPipe, PaCategoryFilterPipe,
        PaDiscountDisplayComponent, PaDiscountEditorComponent,
        PaDiscountPipe, PaDiscountAmountDirective],
    providers: [DiscountService, SimpleDataSource, Model, LogService],
    bootstrap: [ProductComponent]
})
export class AppModule { }
```

Сохранив изменения, вы определите провайдера, необходимого Angular для обработки зависимостей `LogService`. На консоль браузера будут выводиться сообщения следующего вида:

```
Message (INFO): Discount 10 applied to price: 16
```

Для чего нужен этап настройки конфигурации в листинге 20.3? В конце концов, Angular может просто предположить, что новый объект `LogService` должен создаваться в первый раз, когда он потребуется.

Оказывается, Angular предоставляет разных провайдеров, которые создают объекты разными способами, чтобы вы могли управлять процессом создания служб. В табл. 20.3 перечислены разновидности провайдеров, описания которых будут приведены ниже.

Таблица 20.3. Провайдеры Angular

Имя	Описание
Провайдер класса	Провайдер настраивается с использованием класса. Зависимости от службы разрешаются по экземпляру класса, который создается Angular
Провайдер значения	Провайдер настраивается с использованием объекта, который применяется для разрешения зависимостей службы
Провайдер фабрики	Провайдер настраивается с использованием функции. Зависимости от службы разрешаются по объекту, который создается вызовом функции
Провайдер существующей службы	Провайдер настраивается по имени другой службы, что позволяет создавать псевдонимы для служб

Использование провайдера класса

Этот провайдер используется чаще всего. В листинге 20.3 он выбирается добавлением имен классов в свойство `providers` модуля, но это сокращенный синтаксис. Также существует расширенная форма, представленная в листинге 20.4.

Листинг 20.4. Использование расширенного синтаксиса провайдера класса в модуле app.module.ts

```
...
@NgModule({
    imports: [BrowserModule, FormsModule, ReactiveFormsModule],
    declarations: [ProductComponent, PaAttrDirective, PaModel,
        PaStructureDirective, PaIteratorDirective,
        PaCellColor, PaCellColorSwitcher, ProductTableComponent,
        ProductFormComponent, PaAddTaxPipe, PaCategoryFilterPipe,
        PaDiscountDisplayComponent, PaDiscountEditorComponent,
        PaDiscountPipe, PaDiscountAmountDirective],
    providers: [DiscountService, SimpleDataSource, Model,
                { provide: LogService, useClass: LogService }],
    bootstrap: [ProductComponent]
})
...
```

Провайдеры определяются как классы, но их также можно задавать и настраивать в литеральном формате объектов JavaScript:

```
...
{
    provide: LogService,
    useClass: LogService
}
...
```

Провайдер класса поддерживает три свойства, перечисленные в табл. 20.4. Эти свойства подробно описаны ниже.

Таблица 20.4. Свойства провайдера класса

Имя	Описание
provide	Свойство используется для задания маркера (token), предназначенного для идентификации провайдера и разрешения зависимостей. См. раздел «Для чего нужен маркер?»
useClass	Свойство используется для задания класса, экземпляр которого создается провайдером для разрешения зависимости. См. раздел «Свойство useClass»
multi	Свойство используется для передачи массива объектов служб для разрешения зависимостей. См. раздел «Разрешение зависимостей с множественными объектами»

Для чего нужен маркер?

Работа всех провайдеров зависит от *маркера* (token), который используется системой внедрения зависимостей для идентификации зависимости, разрешаемой провайдером. В простейшем варианте в качестве маркера используется класс; этот способ применен в листинге 20.4.

Однако вы можете использовать в качестве маркера любой объект, который позволяет отделить зависимости от типа объекта. Этот механизм повышает гибкость конфигурации внедрения зависимостей, потому что он позволяет провайдеру предоставлять объекты разных типов. Данная возможность может быть полезна для более сложных провайдеров, описанных ниже в этой главе.

В листинге 20.5 провайдер класса используется для регистрации службы записи сообщений в журнал, созданной в начале главы, с использованием строкового маркера.

Листинг 20.5. Регистрация службы с маркером в файле app.module.ts

```
...
@NgModule({
    imports: [BrowserModule, FormsModule, ReactiveFormsModule],
    declarations: [ProductComponent, PaAttrDirective, PaModel,
        PaStructureDirective, PaIteratorDirective,
        PaCellColor, PaCellColorSwitcher, ProductTableComponent,
        ProductFormComponent, PaAddTaxPipe, PaCategoryFilterPipe,
        PaDiscountDisplayComponent, PaDiscountEditorComponent,
        PaDiscountPipe, PaDiscountAmountDirective],
    providers: [DiscountService, SimpleDataSource, Model,
                { provide: "logger", useClass: LogService }],
    bootstrap: [ProductComponent]
})
...
```

В листинге свойству `provide` нового провайдера задается значение `logger`. Angular автоматически сопоставляет провайдеров, маркером которых является класс, но для других типов маркеров потребуется дополнительная помощь. В листинге 20.6 класс `DiscountService` обновляется зависимостью от службы записи сообщений в журнал, для которой используется маркер `logger`.

Листинг 20.6. Использование строкового маркера провайдера в файле discount.service.ts

```
import { Injectable, Inject } from "@angular/core";
import { LogService } from "./log.service";

@Injectable()
export class DiscountService {
    private discountValue: number = 10;

    constructor(@Inject("logger") private logger: LogService) { }
```

```
    public get discount(): number {
        return this.discountValue;
    }

    public set discount(newValue: number) {
        this.discountValue = newValue || 0;
    }

    public applyDiscount(price: number) {
        this.logger.logInfoMessage(`Discount ${this.discount}`
            + ` applied to price: ${price}`);
        return Math.max(price - this.discountValue, 5);
    }
}
```

Декоратор `@Inject` применяется к аргументу конструктора; он определяет маркер провайдера, который будет использоваться для разрешения зависимостей. Когда потребуется создать экземпляр класса `DiscountService`, Angular анализирует конструктор и использует аргумент декоратора `@Inject` для выбора провайдера, который будет использоваться для разрешения зависимостей.

Класс OpaqueToken

Когда в качестве маркера провайдера используются простые типы, появляется возможность того, что две разные части приложения попробуют использовать один маркер для идентификации разных служб. В результате для разрешения зависимостей может быть использован неверный тип, что приведет к ошибке.

Для решения этой проблемы Angular предоставляет класс `OpaqueToken`, предоставляющий объектную обертку для строкового значения; он может использоваться для создания уникальных значений маркеров. В листинге 20.7 я использовал класс `OpaqueToken` для создания маркера, предназначенного для идентификации зависимостей от класса `LogService`.

Листинг 20.7. Создание маркера с использованием класса OpaqueToken в файле log.service.ts

```
import { Injectable, OpaqueToken } from "@angular/core";
export const LOG_SERVICE = new OpaqueToken("logger");

export enum LogLevel {
    DEBUG, INFO, ERROR
}

@Injectable()
export class LogService {
    minimumLevel: LogLevel = LogLevel.INFO;

    // ...Методы опущены для краткости...
}
```

Конструктор класса `OpaqueToken` получает строковое значение, которое идентифицирует службу, но маркером при этом становится объект `OpaqueToken`. Зависимости должны объявляться для того же объекта `OpaqueToken`, который использовался для создания провайдера в модуле; именно поэтому маркер создается с ключевым словом `const`, предотвращающим модификацию объекта. В листинге 20.8 показана конфигурация провайдера с новым маркером.

Листинг 20.8. Создание провайдера с использованием OpaqueToken в файле app.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { ProductComponent } from "./component";
import { FormsModule, ReactiveFormsModule  } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
import { PaModel } from "./twoway.directive";
import { PaStructureDirective } from "./structure.directive";
import { PaIteratorDirective } from "./iterator.directive";
import { PaCellColor } from "./cellColor.directive";
import { PaCellColorSwitcher } from "./cellColorSwitcher.directive";
import { ProductTableComponent } from "./productTable.component";
import { ProductFormComponent } from "./productForm.component";
import { PaAddTaxPipe } from "./addTax.pipe";
import { PaCategoryFilterPipe } from "./categoryFilter.pipe";
import { PaDiscountDisplayComponent } from "./discountDisplay.component";
import { PaDiscountEditorComponent } from "./discountEditor.component";
import { DiscountService } from "./discount.service";
import { PaDiscountPipe } from "./discount.pipe";
import { PaDiscountAmountDirective } from "./discountAmount.directive";
import { SimpleDataSource } from "./datasource.model";
import { Model } from "./repository.model";
import { LogService, LOG_SERVICE } from "./log.service";

@NgModule({
    imports: [BrowserModule, FormsModule, ReactiveFormsModule],
    declarations: [ProductComponent, PaAttrDirective, PaModel,
        PaStructureDirective, PaIteratorDirective,
        PaCellColor, PaCellColorSwitcher, ProductTableComponent,
        ProductFormComponent, PaAddTaxPipe, PaCategoryFilterPipe,
        PaDiscountDisplayComponent, PaDiscountEditorComponent,
        PaDiscountPipe, PaDiscountAmountDirective],
    providers: [DiscountService, SimpleDataSource, Model,
        { provide: LOG_SERVICE, useClass: LogService }],
    bootstrap: [ProductComponent]
})
export class AppModule { }
```

Наконец, в листинге 20.9 показана обновленная версия класса `DiscountService`, в которой зависимость объявляется с использованием `OpaqueToken` вместо `string`.

Листинг 20.9. Объявление зависимости с OpaqueToken в файле discount.service.ts

```
import { Injectable, Inject } from "@angular/core";
import { LogService, LOG_SERVICE } from "./log.service";

@Injectable()
export class DiscountService {
    private discountValue: number = 10;

    constructor( @Inject(LOG_SERVICE) private logger: LogService) { }

    public get discount(): number {
        return this.discountValue;
    }

    public set discount(newValue: number) {
        this.discountValue = newValue || 0;
    }

    public applyDiscount(price: number) {
        this.logger.logInfoMessage(`Discount ${this.discount}`
            + ` applied to price: ${price}`);
        return Math.max(price - this.discountValue, 5);
    }
}
```

В функциональности приложения ничего не изменилось, но использование `OpaqueToken` предотвращает возможные конфликты служб и путаницу.

Свойство useClass

Свойство `useClass` провайдера класса задает класс, экземпляр которого будет создаваться для разрешения зависимостей. Провайдер может быть настроен с любым классом; это означает, что реализацию службы можно сменить, изменив конфигурацию провайдера. Тем не менее этой возможностью следует пользоваться с осторожностью, потому что получатели объекта службы будут ожидать конкретный тип, причем несоответствия не создадут проблем вплоть до запуска приложения в браузере (система контроля типов TypeScript не влияет на внедрение зависимостей, потому что оно происходит на стадии выполнения — уже после того, как аннотации типов будут обработаны компилятором TypeScript).

Наиболее распространенный способ изменения классов — использование других субклассов. В листинге 20.10 класс `LogService` расширяется с созданием службы, выводящей на консоль JavaScript сообщения в другом формате.

Листинг 20.10. Создание субклассированной службы в файле log.service.ts

```
import { Injectable, OpaqueToken } from "@angular/core";
export const LOG_SERVICE = new OpaqueToken("logger");
```

```
export enum LogLevel {
    DEBUG, INFO, ERROR
}

@Injectable()
export class LogService {
    minimumLevel: LogLevel = LogLevel.INFO;

    logInfoMessage(message: string) {
        this.logMessage(LogLevel.INFO, message);
    }

    logDebugMessage(message: string) {
        this.logMessage(LogLevel.DEBUG, message);
    }

    logErrorMessage(message: string) {
        this.logMessage(LogLevel.ERROR, message);
    }

    logMessage(level: LogLevel, message: string) {
        if (level >= this.minimumLevel) {
            console.log(`Message (${LogLevel[level]}): ${message}`);
        }
    }
}

@Injectable()
export class SpecialLogService extends LogService {

    constructor() {
        super()
        this.minimumLevel = LogLevel.DEBUG;
    }

    logMessage(level: LogLevel, message: string) {
        if (level >= this.minimumLevel) {
            console.log(`Special Message (${LogLevel[level]}): ${message}`);
        }
    }
}
```

Класс `SpecialLogService` расширяет `LogService` и предоставляет собственную реализацию метода `logMessage`. В листинге 20.11 конфигурация провайдера обновляется с назначением новой службы в свойстве `useClass`.

Листинг 20.11. Настройка конфигурации провайдера в файле app.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { ProductComponent } from "./component";
import { FormsModule, ReactiveFormsModule  } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
```

```
import { PaModel } from "./twoway.directive";
import { PaStructureDirective } from "./structure.directive";
import { PaIteratorDirective } from "./iterator.directive";
import { PaCellColor } from "./cellColor.directive";
import { PaCellColorSwitcher } from "./cellColorSwitcher.directive";
import { ProductTableComponent } from "./productTable.component";
import { ProductFormComponent } from "./productForm.component";
import { PaAddTaxPipe } from "./addTax.pipe";
import { PaCategoryFilterPipe } from "./categoryFilter.pipe";
import { PaDiscountDisplayComponent } from "./discountDisplay.component";
import { PaDiscountEditorComponent } from "./discountEditor.component";
import { DiscountService } from "./discount.service";
import { PaDiscountPipe } from "./discount.pipe";
import { PaDiscountAmountDirective } from "./discountAmount.directive";
import { SimpleDataSource } from "./datasource.model";
import { Model } from "./repository.model";
import { LogService, LOG_SERVICE, SpecialLogService } from "./log.service";

@NgModule({
    imports: [BrowserModule, FormsModule, ReactiveFormsModule],
    declarations: [ProductComponent, PaAttrDirective, PaModel,
        PaStructureDirective, PaIteratorDirective,
        PaCellColor, PaCellColorSwitcher, ProductTableComponent,
        ProductFormComponent, PaAddTaxPipe, PaCategoryFilterPipe,
        PaDiscountDisplayComponent, PaDiscountEditorComponent,
        PaDiscountPipe, PaDiscountAmountDirective],
    providers: [DiscountService, SimpleDataSource, Model,
        { provide: LOG_SERVICE, useClass: SpecialLogService }],
    bootstrap: [ProductComponent]
})
export class AppModule { }
```

Комбинация маркера и класса означает, что зависимости от маркера `LOG_SERVICE` будут разрешаться с использованием объекта `SpecialLogService`. Сохранив изменения, вы увидите на консоли сообщения, свидетельствующие о том, что используется производная служба:

```
Special Message (INFO): Discount 10 applied to price: 275
```

Будьте осторожны при использовании свойства `useClass` для задания типа, который ожидают получить зависимые классы. Передача субкласса — безопасный вариант, потому что в субклассе гарантированно будет доступна функциональность базового класса.

Разрешение зависимостей с множественными объектами

Провайдер класса можно настроить на передачу массива объектов для разрешения зависимости. Например, эта возможность может быть полезна, если вы хотите предоставить набор взаимосвязанных служб, отличающихся настройкой конфигурации. Чтобы предоставить массив, настройте несколько провайдеров класса с использованием одного маркера и задайте свойству `multi` значение `true` (листинг 20.12).

Листинг 20.12. Настройка множественных объектов служб в файле app.module.ts

```
...
@NgModule({
    imports: [BrowserModule, FormsModule, ReactiveFormsModule],
    declarations: [ProductComponent, PaAttrDirective, PaModel,
        PaStructureDirective, PaIteratorDirective,
        PaCellColor, PaCellColorSwitcher, ProductTableComponent,
        ProductFormComponent, PaAddTaxPipe, PaCategoryFilterPipe,
        PaDiscountDisplayComponent, PaDiscountEditorComponent,
        PaDiscountPipe, PaDiscountAmountDirective],
    providers: [DiscountService, SimpleDataSource, Model,
        { provide: LOG_SERVICE, useClass: LogService, multi: true },
        { provide: LOG_SERVICE, useClass: SpecialLogService, multi: true }],
    bootstrap: [ProductComponent]
})
...
```

Чтобы разрешать зависимости от маркера `LOG_SERVICE`, система внедрения зависимостей Angular создает объекты `LogService` и `SpecialLogService`, помещает их в массив и передает конструктору зависимого класса. Класс, получающий службы, должен ожидать, что ему будет передан массив (листинг 20.13).

Листинг 20.13. Получение множественных служб в файле discount.service.ts

```
import { Injectable, Inject } from "@angular/core";
import { LogService, LOG_SERVICE, LogLevel } from "./log.service";

@Injectable()
export class DiscountService {
    private discountValue: number = 10;
    private logger: LogService;

    constructor( @Inject(LOG_SERVICE) loggers: LogService[]) {
        this.logger = loggers.find(l => l.minimumLevel == LogLevel.DEBUG);
    }

    public get discount(): number {
        return this.discountValue;
    }

    public set discount(newValue: number) {
        this.discountValue = newValue || 0;
    }

    public applyDiscount(price: number) {
        this.logger.logInfoMessage(`Discount ${this.discount}`
            + ` applied to price: ${price}`);
        return Math.max(price - this.discountValue, 5);
    }
}
```

Конструктор получает службы в массиве, использует метод `find` массива для обнаружения первого объекта, у которого свойство `minimumLevel` содержит

LogLevel.Debug, и задает его свойству logger. Метод applyDiscount вызывает метод logDebugMessage службы; в результате на консоль JavaScript выводятся сообщения следующего вида:

```
Special Message (INFO): Discount 10 applied to price: 275
```

Использование провайдера значения

Провайдер значения используется в тех ситуациях, когда вы хотите взять ответственность за создание объектов служб на себя, не оставляя ее на провайдере класса. Также данная возможность может быть полезна, если службы являются простыми типами (например, значениями string или number); провайдер значений становится удобным способом получения доступа к часто используемым настройкам конфигурации.

Провайдер значения может применяться с использованием объектного литерала. Поддерживаемые им свойства перечислены в табл. 20.5.

Таблица 20.5. Свойства провайдера значения

Имя	Описание
provide	Свойство определяет маркер службы (см. раздел «Для чего нужен маркер?» этой главы)
useValue	Свойство задает объект, который будет использоваться для разрешения зависимостей
multi	Свойство используется для объединения нескольких провайдеров в массиве объектов, который будет использоваться для разрешения зависимости от маркера. См. раздел «Разрешение зависимостей с множественными объектами»

Провайдер значения работает так же, как провайдер класса, не считая того, что для настройки его конфигурации используется объект, а не тип. В листинге 20.14 продемонстрировано использование провайдера значения для создания экземпляра класса LogService, который инициализируется с конкретным значением свойства.

Листинг 20.14. Использование провайдера значения в файле app.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { ProductComponent } from "./component";
import { FormsModule, ReactiveFormsModule  } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
import { PaModel } from "./twoway.directive";
import { PaStructureDirective } from "./structure.directive";
import { PaIteratorDirective } from "./iterator.directive";
import { PaCellColor } from "./cellColor.directive";
import { PaCellColorSwitcher } from "./cellColorSwitcher.directive";
import { ProductTableComponent } from "./productTable.component";
import { ProductFormComponent } from "./productForm.component";
```

```
import { PaAddTaxPipe } from "./addTax.pipe";
import { PaCategoryFilterPipe } from "./categoryFilter.pipe";
import { PaDiscountDisplayComponent } from "./discountDisplay.component";
import { PaDiscountEditorComponent } from "./discountEditor.component";
import { DiscountService } from "./discount.service";
import { PaDiscountPipe } from "./discount.pipe";
import { PaDiscountAmountDirective } from "./discountAmount.directive";
import { SimpleDataSource } from "./datasource.model";
import { Model } from "./repository.model";
import { LogService, LOG_SERVICE, SpecialLogService, LogLevel } from "./log.
service";

let logger = new LogService();
logger.minimumLevel = LogLevel.DEBUG;

@NgModule({
    imports: [BrowserModule, FormsModule, ReactiveFormsModule],
    declarations: [ProductComponent, PaAttrDirective, PaModel,
        PaStructureDirective, PaIteratorDirective,
        PaCellColor, PaCellColorSwitcher, ProductTableComponent,
        ProductFormComponent, PaAddTaxPipe, PaCategoryFilterPipe,
        PaDiscountDisplayComponent, PaDiscountEditorComponent,
        PaDiscountPipe, PaDiscountAmountDirective],
    providers: [DiscountService, SimpleDataSource, Model,
        { provide: LogService, useValue: logger }],
    bootstrap: [ProductComponent]
})
export class AppModule { }
```

Этот провайдер значения настраивается для разрешения зависимостей от маркера `LogService` с конкретным объектом, который был создан и настроен за пределами класса модуля.

Провайдер значения — как, собственно, и все провайдеры — может использовать в качестве маркера любой объект (см. предыдущий раздел), но я вернулся к использованию типов в качестве маркеров, потому что этот способ чаще всего встречается на практике и потому что он хорошо сочетается с типизацией параметров конструктора TypeScript. В листинге 20.15 представлены соответствующие изменения в классе `DiscountService`, который объявляет зависимость с использованием типизованного аргумента конструктора.

Листинг 20.15. Объявление зависимости с использованием типа в файле discount.service.ts

```
import { Injectable, Inject } from "@angular/core";
import { LogService, LOG_SERVICE, LogLevel } from "./log.service";

@Injectable()
export class DiscountService {
    private discountValue: number = 10;

    constructor(private logger: LogService) { }
```

```
    public get discount(): number {
        return this.discountValue;
    }

    public set discount(newValue: number) {
        this.discountValue = newValue || 0;
    }

    public applyDiscount(price: number) {
        this.logger.logInfoMessage(`Discount ${this.discount}`
            + ` applied to price: ${price}`);
        return Math.max(price - this.discountValue, 5);
    }
}
```

Использование провайдера фабрики

Провайдер фабрики использует функцию для создания объекта, необходимого для разрешения зависимости. Этот провайдер поддерживает свойства, перечисленные в табл. 20.6.

Таблица 20.6. Свойства провайдера фабрики

Имя	Описание
provide	Свойство определяет маркер службы (см. раздел «Для чего нужен маркер?» этой главы)
deps	Свойство задает массив маркеров провайдеров, которые будут разрешены и переданы функции, заданной свойством useFactory
useFactory	Свойство задает функцию, которая создает объект службы. Объекты, полученные в результате разрешения маркеров, заданных свойством deps, будут переданы функции в аргументах. Результат, возвращенный функцией, используется как объект службы
multi	Свойство используется для объединения нескольких провайдеров в массиве объектов, который будет использоваться для разрешения зависимости от маркера. См. раздел «Разрешение зависимостей с множественными объектами»

Этот провайдер обеспечивает наибольшую гибкость в создании объектов служб, потому что вы можете определять функции, приспособленные к требованиям вашего приложения. В листинге 20.16 приведена фабричная функция для создания объектов `LogService`.

Листинг 20.16. Использование провайдера фабрики для файла app.module.ts

```
...
@NgModule({
    imports: [BrowserModule, FormsModule, ReactiveFormsModule],
    declarations: [ProductComponent, PaAttrDirective, PaModel,
```

```
        PaStructureDirective, PaIteratorDirective,
        PaCellColor, PaCellColorSwitcher, ProductTableComponent,
        ProductFormComponent, PaAddTaxPipe, PaCategoryFilterPipe,
        PaDiscountDisplayComponent, PaDiscountEditorComponent,
        PaDiscountPipe, PaDiscountAmountDirective],
    providers: [DiscountService, SimpleDataSource, Model,
        {
            provide: LogService, useFactory: () => {
                let logger = new LogService();
                logger.minimumLevel = LogLevel.DEBUG;
                return logger;
            }
        }],

    bootstrap: [ProductComponent]
})
```

Функция в этом примере проста: она не получает аргументов и просто создает новый объект `LogService`. По-настоящему гибкость этого провайдера проявляется при использовании свойства `deps`, позволяющего создавать зависимости от других служб. В листинге 20.17 определяется маркер, задающий уровень регистрации для отладки.

Листинг 20.17. Определение службы уровня регистрации в файле log.service.ts

```
import { Injectable, OpaqueToken } from "@angular/core";

export const LOG_SERVICE = new OpaqueToken("logger");
export const LOG_LEVEL = new OpaqueToken("log_level");

export enum LogLevel {
    DEBUG, INFO, ERROR
}

@Injectable()
export class LogService {
    minimumLevel: LogLevel = LogLevel.INFO;

    // ...Методы опущены для краткости...

}

@Injectable()
export class SpecialLogService extends LogService {

    // ...Методы опущены для краткости...
}
```

В листинге 20.18 определяется провайдер значения, который создает службу с маркером `LOG_LEVEL`. Служба используется в фабричной функции, которая создает объект `LogService`.

Листинг 20.18. Использование фабричных зависимостей в файле app.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { ProductComponent } from "./component";
import { FormsModule, ReactiveFormsModule  } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
import { PaModel } from "./twoway.directive";
import { PaStructureDirective } from "./structure.directive";
import { PaIteratorDirective } from "./iterator.directive";
import { PaCellColor } from "./cellColor.directive";
import { PaCellColorSwitcher } from "./cellColorSwitcher.directive";
import { ProductTableComponent } from "./productTable.component";
import { ProductFormComponent } from "./productForm.component";
import { PaAddTaxPipe } from "./addTax.pipe";
import { PaCategoryFilterPipe } from "./categoryFilter.pipe";
import { PaDiscountDisplayComponent } from "./discountDisplay.component";
import { PaDiscountEditorComponent } from "./discountEditor.component";
import { DiscountService } from "./discount.service";
import { PaDiscountPipe } from "./discount.pipe";
import { PaDiscountAmountDirective } from "./discountAmount.directive";
import { SimpleDataSource } from "./datasource.model";
import { Model } from "./repository.model";
import { LogService, LOG_SERVICE, SpecialLogService,
         LogLevel, LOG_LEVEL} from "./log.service";
@NgModule({
    imports: [BrowserModule, FormsModule, ReactiveFormsModule],
    declarations: [ProductComponent, PaAttrDirective, PaModel,
        PaStructureDirective, PaIteratorDirective,
        PaCellColor, PaCellColorSwitcher, ProductTableComponent,
        ProductFormComponent, PaAddTaxPipe, PaCategoryFilterPipe,
        PaDiscountDisplayComponent, PaDiscountEditorComponent,
        PaDiscountPipe, PaDiscountAmountDirective],
    providers: [DiscountService, SimpleDataSource, Model,
        { provide: LOG_LEVEL, useValue: LogLevel.DEBUG },
        { provide: LogService,
          deps: [LOG_LEVEL],
          useFactory: (level) => {
            let logger = new LogService();
            logger.minimumLevel = level;
            return logger;
         }
        }],
    bootstrap: [ProductComponent]
})
export class AppModule { }
```

Маркер `LOG_LEVEL` используется провайдером значения для определения простого значения в качестве службы. Провайдер фабрики задает этот маркер в своем массиве `deps`; система внедрения зависимостей разрешает его и передает в аргументе фабричной функции, где используется для задания свойства `minimumLevel` нового объекта `LogService`.

Использование провайдера существующей службы

Этот провайдер используется для создания псевдонимов служб, чтобы к ним можно было обращаться по разным маркерам. Его свойства перечислены в табл. 20.7.

Таблица 20.7. Свойства провайдера существующей службы

Имя	Описание
provide	Свойство определяет маркер службы. См. раздел «Для чего нужен маркер?»
useExisting	Свойство используется для задания маркера другого провайдера, объект службы которого может использоваться для разрешения зависимостей этой службы
multi	Свойство используется для передачи массива объектов служб для разрешения зависимостей. См. раздел «Разрешение зависимостей с множественными объектами»

Этот провайдер может использоваться в том случае, если вы хотите переработать набор провайдеров, но не хотите удалять все устаревшие маркеры, чтобы избежать переработки остального кода приложения. Пример использования провайдера приведен в листинге 20.19.

Листинг 20.19. Создание псевдонима службы в файле app.module.ts

```
...
@NgModule({
    imports: [BrowserModule, FormsModule, ReactiveFormsModule],
    declarations: [ProductComponent, PaAttrDirective, PaModel,
        PaStructureDirective, PaIteratorDirective,
        PaCellColor, PaCellColorSwitcher, ProductTableComponent,
        ProductFormComponent, PaAddTaxPipe, PaCategoryFilterPipe,
        PaDiscountDisplayComponent, PaDiscountEditorComponent,
        PaDiscountPipe, PaDiscountAmountDirective],
    providers: [DiscountService, SimpleDataSource, Model,
        { provide: LOG_LEVEL, useValue: LogLevel.DEBUG },
        { provide: "debugLevel", useExisting: LOG_LEVEL },
        { provide: LogService,
          deps: ["debugLevel"],
          useFactory: (level) => {
            let logger = new LogService();
            logger.minimumLevel = level;
            return logger;
         }
        }],
    bootstrap: [ProductComponent]
})
...
```

Маркером новой службы является строка `debugLevel`, которая назначается псевдонимом для провайдера с маркером `LOG_LEVEL`. При использовании любого маркера зависимость будет разрешена с одним и тем же значением.

Использование локальных провайдеров

Создавая новый экземпляр класса, Angular разрешает любые зависимости с использованием *инжектора* (injector). Именно инжектор анализирует конструкторы классов для определения того, какие зависимости были объявлены, и разрешает их с использованием доступных провайдеров.

До настоящего момента все примеры внедрения зависимостей полагались на провайдеров, настроенных в модуле Angular приложения. Однако система внедрения зависимостей Angular устроена сложнее: существует иерархия инжекторов, соответствующая дереву компонентов и директив в приложении. Каждый компонент и каждая директива обладают собственным инжектором, и для каждого инжектора можно настроить собственный набор провайдеров, называемых *локальными провайдерами*.

Столкнувшись с необходимостью разрешения зависимости, Angular использует инжектор ближайшего компонента или директивы. Инжектор сначала пытается разрешить зависимость с использованием собственного набора локальных провайдеров. Если локальные провайдеры не настроены или не существует ни одного провайдера, который мог бы использоваться для разрешения этой конкретной зависимости, то инжектор обращается к инжектору родительского компонента. Процесс повторяется: инжектор родительского компонента пытается разрешить зависимость с использованием своего набора локальных провайдеров. Если подходящий провайдер будет найден, то он используется для получения объекта службы, необходимого для разрешения зависимостей. Если подходящего провайдера нет, то запрос передается наверх на следующий уровень иерархии — прародителю исходного инжектора. На верхнем уровне иерархии располагается корневой модуль Angular, провайдеры которого становятся последней мерой перед выдачей сообщения об ошибке.

Определение провайдеров в модуле Angular означает, что все зависимости маркера в приложении будут разрешаться с использованием одного объекта. Как я объясню в следующих разделах, регистрация провайдеров на нижних уровнях иерархии инжекторов может изменить это поведение и механизм создания/использования служб.

Ограничения модели с одним объектом службы

Использование одного объекта службы может стать эффективным механизмом совместного использования данных в разных частях приложения и обработки взаимодействий с пользователем. Но некоторые службы плохо подходят для столь широкого совместного использования. Рассмотрим простой пример: в листинге 20.20 зависимость от `LogService` добавляется в один из каналов, созданных в главе 18.

Листинг 20.20. Добавление зависимости от службы в файл discount.pipe.ts

```
import { Pipe, Injectable } from "@angular/core";
import { DiscountService } from "./discount.service";
import { LogService } from "./log.service";
```

```
@Pipe({
    name: "discount",
    pure: false
})
export class PaDiscountPipe {

    constructor(private discount: DiscountService,
                private logger: LogService) { }

    transform(price: number): number {
        if (price > 100) {
            this.logger.logInfoMessage(`Large price discounted: ${price}`);
        }
        return this.discount.applyDiscount(price);
    }
}
```

Метод `transform` канала использует объект `LogService`, передаваемый в аргументе конструктора, для генерирования журнальных сообщений, если преобразуемая цена больше 100.

Проблема в том, что эти сообщения теряются в потоке сообщений, генерируемых объектом `DiscountService`, который создает сообщение для каждого применения скидки. Первое, что приходит в голову, — изменение минимального уровня в объекте `LogService`, когда он создается фабричной функцией провайдера модуля (листинг 20.21).

Листинг 20.21. Изменение уровня регистрации в файле app.module.ts

```
...
@NgModule({
    imports: [BrowserModule, FormsModule, ReactiveFormsModule],
    declarations: [ProductComponent, PaAttrDirective, PaModel,
        PaStructureDirective, PaIteratorDirective,
        PaCellColor, PaCellColorSwitcher, ProductTableComponent,
        ProductFormComponent, PaAddTaxPipe, PaCategoryFilterPipe,
        PaDiscountDisplayComponent, PaDiscountEditorComponent,
        PaDiscountPipe, PaDiscountAmountDirective],
    providers: [DiscountService, SimpleDataSource, Model,
        { provide: LOG_LEVEL, useValue: LogLevel.ERROR },
        { provide: "debugLevel", useExisting: LOG_LEVEL },
        {
            provide: LogService,
            deps: ["debugLevel"],
            useFactory: (level) => {
                let logger = new LogService();
                logger.minimumLevel = level;
                return logger;
            }
        }],
    bootstrap: [ProductComponent]
})
...
```

Конечно, это не приводит к желаемому эффекту, потому что один объект `LogService` используется во всем приложении и фильтрация сообщений `DiscountService` означает, что сообщения канала тоже будут отфильтрованы.

Можно было бы доработать класс `LogService`, чтобы для каждого источника сообщений создавался свой фильтр, но такое решение быстро усложняется. Вместо этого проблема будет решаться созданием локального провайдера, чтобы в приложении существовало несколько объектов `LogService`, каждый из которых настраивается отдельно от других.

Создание локальных провайдеров в директиве

Директивы определяют локальных провайдеров в свойстве `providers` с использованием провайдеров и синтаксиса, описанных в начале главы. Когда Angular потребуется разрешить зависимость директивы или одного из ее контентных потомков (директивы или каналы, содержащиеся в управляющем элементе), будет использован локальный провайдер. В листинге 20.22 показан локальный провайдер, созданный директивой.

Листинг 20.22. Создание локального провайдера в файле cellColorSwitches.directive.ts

```
import {
    Directive, Input, Output, EventEmitter,
    SimpleChange, ContentChildren, QueryList
} from "@angular/core";
import { PaCellColor } from "./cellColor.directive";
import { LogService } from "./log.service";

@Directive({
    selector: "table",
    providers: [LogService]
})
export class PaCellColorSwitcher {

    @Input("paCellDarkColor")
    modelProperty: Boolean;

    @ContentChildren(PaCellColor)
    contentChildren: QueryList<PaCellColor>;

    ngOnChanges(changes: { [property: string]: SimpleChange }) {
        this.updateContentChildren(changes["modelProperty"].currentValue);
    }

    ngAfterContentInit() {
        this.contentChildren.changes.subscribe(() => {
            setTimeout(() => this.updateContentChildren(this.modelProperty), 0);
        });
    }

    private updateContentChildren(dark: Boolean) {
        if (this.contentChildren != null && dark != undefined) {
```

```
            this.contentChildren.forEach((child, index) => {
                child.setColor(index % 2 ? dark : !dark);
            });
        }
    }
}
```

В предшествующей главе эта директива отвечала за изменение цвета ячеек таблицы, для чего она координировала свою работу с другой директивой. Может показаться, что это странное место для создания локального провайдера, но иерархия инжекторов следует структуре приложения, а эта директива работает с элементом `table`; это означает, что объекты каналов с зависимостью `LogService` являются дочерними по отношению к управляющему элементу директивы.

Когда потребуется создать новый объект канала, Angular обнаруживает зависимость от `LogService`, начинает подниматься по иерархии приложения, проверять каждый компонент и директиву и определять, есть ли у них провайдер, который может использоваться для разрешения зависимости. Директива `PaCellColorSwitcher` содержит провайдер `LogService`, который использовался для создания службы, задействованной в разрешении зависимости канала. Это означает, что в приложении есть два объекта `LogService`, которые могут настраиваться по отдельности (рис. 20.2).

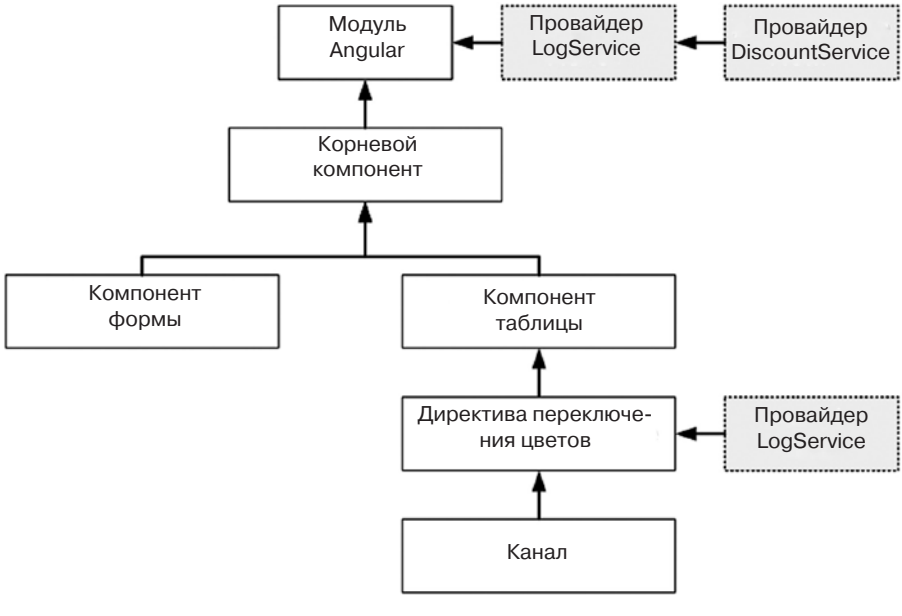


Рис. 20.2. Создание локального провайдера

Объект `LogService`, созданный провайдером директивы переключения цветов, использует значение по умолчанию для свойства `minimumLevel` и выводит сообщения

LogLevel.INFO. Созданный модулем объект LogService, который будет использоваться для разрешения всех остальных зависимостей в приложении (включая объявленную классом DiscountService), настраивается для вывода только сообщений LogLevel.ERROR. После сохранения изменений вы будете видеть сообщения от канала (получающего службу от директивы), но не от DiscountService (получающего службу от модуля).

Создание локальных провайдеров в компонентах

Создавать локальные провайдеры в компонентах сложнее, потому что разработчику приходится принимать решение относительно видимости созданных служб. Компоненты поддерживают два свойства декораторов для создания локальных провайдеров (табл. 20.8).

Таблица 20.8. Свойства декоратора компонента локального провайдера

Имя	Описание
providers	Свойство используется для создания провайдера, используемого для разрешения потомков представлений и контентных потомков
viewProviders	Свойство используется для создания провайдера, используемого для разрешения зависимостей потомков представлений

Чтобы продемонстрировать применение этих свойств, создайте файл valueDisplay.directive.ts в папке app и включите в него определение директивы из листинга 20.23.

Листинг 20.23. Содержимое файла valueDisplay.directive.ts в папке app

```
import { Directive, OpaqueToken, Inject, HostBinding} from "@angular/core";
export const VALUE_SERVICE = new OpaqueToken("value_service");

@Directive({
    selector: "[paDisplayValue]"
})
export class PaDisplayValueDirective {

    constructor( @Inject(VALUE_SERVICE) serviceValue: string) {
        this.elementContent = serviceValue;
    }

    @HostBinding("textContent")
    elementContent: string;
}
```

Маркер VALUE_SERVICE используется для определения службы на базе значения. Директива в листинге объявляет зависимость от этой службы для отображения в контенте управляющего элемента. В листинге 20.24 продемонстрировано определение службы и регистрация директивы в модуле Angular. Я также упростил провайдер LogService в модуле для экономии места.

Листинг 20.24. Регистрация директивы и службы в файле app.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { ProductComponent } from "./component";
import { FormsModule, ReactiveFormsModule  } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
import { PaModel } from "./twoway.directive";
import { PaStructureDirective } from "./structure.directive";
import { PaIteratorDirective } from "./iterator.directive";
import { PaCellColor } from "./cellColor.directive";
import { PaCellColorSwitcher } from "./cellColorSwitcher.directive";
import { ProductTableComponent } from "./productTable.component";
import { ProductFormComponent } from "./productForm.component";
import { PaAddTaxPipe } from "./addTax.pipe";
import { PaCategoryFilterPipe } from "./categoryFilter.pipe";
import { PaDiscountDisplayComponent } from "./discountDisplay.component";
import { PaDiscountEditorComponent } from "./discountEditor.component";
import { DiscountService } from "./discount.service";
import { PaDiscountPipe } from "./discount.pipe";
import { PaDiscountAmountDirective } from "./discountAmount.directive";
import { SimpleDataSource } from "./datasource.model";
import { Model } from "./repository.model";
import { LogService, LOG_SERVICE, SpecialLogService,
    LogLevel, LOG_LEVEL} from "./log.service";
import { VALUE_SERVICE, PaDisplayValueDirective} from "./valueDisplay.directive";

@NgModule({
    imports: [BrowserModule, FormsModule, ReactiveFormsModule],
    declarations: [ProductComponent, PaAttrDirective, PaModel,
        PaStructureDirective, PaIteratorDirective,
        PaCellColor, PaCellColorSwitcher, ProductTableComponent,
        ProductFormComponent, PaAddTaxPipe, PaCategoryFilterPipe,
        PaDiscountDisplayComponent, PaDiscountEditorComponent,
        PaDiscountPipe, PaDiscountAmountDirective, PaDisplayValueDirective],
    providers: [DiscountService, SimpleDataSource, Model, LogService,
                { provide: VALUE_SERVICE, useValue: "Apples" }],
    bootstrap: [ProductComponent]
})
export class AppModule { }
```

Провайдер задает значение `Apples` для службы `VALUE_SERVICE`. На следующем шаге новая директива применяется таким образом, что появляются два экземпляра: потомок представления компонента и контентный потомок.

В листинге 20.25 создается экземпляр контентного потомка.

Листинг 20.25. Применение директивы контентного потомка в файле template.html

```
<div class="col-xs-4 p-a-1">
    <paProductForm>
        <span paDisplayValue></span>
    </paProductForm>
</div>
<div class="col-xs-8 p-a-1">
    <paProductTable></paProductTable>
</div>
```

Листинг 20.26 проецирует контент управляющего элемента и добавляет экземпляр потомка представления в новую директиву.

Листинг 20.26. Добавление директив в файл productForm.component.html

```
<form novalidate [formGroup]="form" (ngSubmit)="submitForm(form)">
    <div class="form-group" *ngFor="let control of form.productControls">
        <label>{{control.label}}</label>
        <input class="form-control"
            [(ngModel)]="newProduct[control.modelProperty]"
            name="{{control.modelProperty}}"
            formControlName="{{control.modelProperty}}" />
        <ul class="text-danger list-unstyled"
                *ngIf="(formSubmitted || control.dirty) && !control.valid">
            <li *ngFor="let error of control.getValidationMessages()">
                {{error}}
            </li>
        </ul>
    </div>
    <button class="btn btn-primary" type="submit"
        [disabled]="formSubmitted && !form.valid"
        [class.btn-secondary]="formSubmitted && !form.valid">
            Create
    </button>
</form>
<div class="bg-info m-t-1 p-a-1">
    View Child Value: <span paDisplayValue></span>
</div>
<div class="bg-info m-t-1 p-a-1">
    Content Child Value: <ng-content></ng-content>
</div>
```

Сохранив изменения, вы увидите новые элементы (рис. 20.3) с одинаковыми значениями, потому что в модуле определяется всего один провайдер для VALUE_SERVICE.

Рис. 20.3. Директивы потомка представления и контентного потомка

Создание локального провайдера для всех потомков

Свойство `providers` декоратора `@Component` используется для определения провайдеров, которые будут использоваться для разрешения зависимостей служб для всех потомков — как определяемых в шаблоне (потомки представления), так и проецируемых из управляющего элемента (контентные потомки). В листинге 20.27 провайдер `VALUE_SERVICE` определяется в родительском компоненте для двух новых экземпляров директивы.

Листинг 20.27. Определение провайдера в файле productForm.component.ts

```
import { Component, Output, EventEmitter, ViewEncapsulation } from "@angular/core";
import { Product } from "./product.model";
import { ProductFormGroup } from "./form.model";
import { Model } from "./repository.model";
import { VALUE_SERVICE } from "./valueDisplay.directive";

@Component({
    selector: "paProductForm",
    templateUrl: "app/productForm.component.html",
    providers: [{ provide: VALUE_SERVICE, useValue: "Oranges" }]
})
export class ProductFormComponent {
    form: ProductFormGroup = new ProductFormGroup();
    newProduct: Product = new Product();
    formSubmitted: boolean = false;

    constructor(private model: Model) { }

    submitForm(form: any) {
        this.formSubmitted = true;
        if (form.valid) {
            this.model.saveProduct(this.newProduct);
            this.newProduct = new Product();
            this.form.reset();
            this.formSubmitted = false;
        }
    }
}
```

Новый провайдер изменяет значение службы. Когда доходит до создания экземпляров новой директивы, Angular начинает поиск провайдеров вверх по иерархии приложения и находит провайдера `VALUE_SERVICE`, определенного в листинге 20.27. Значение службы используется обоими экземплярами директивы (рис. 20.4).

Создание провайдера для потомков представлений

Свойство `viewProviders` определяет провайдеров, которые используются для разрешения зависимостей для потомков представлений, но не контентных потомков. В листинге 20.28 свойство `viewProviders` используется для определения провайдера для `VALUE_SERVICE`.

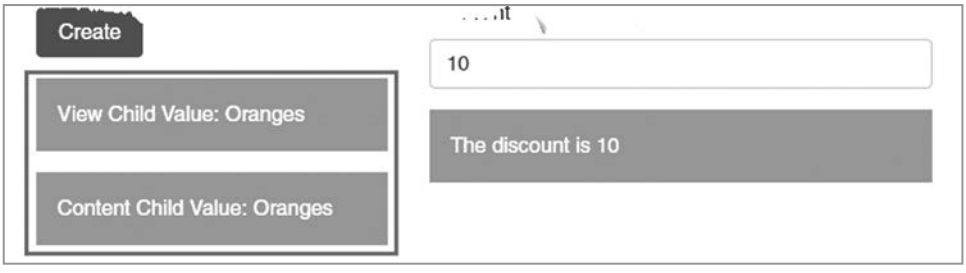


Рис. 20.4. Определение провайдера для всех потомков в компоненте

Листинг 20.28. Определение провайдера потомка представления в файле productForm.component.ts

```
import { Component, Output, EventEmitter, ViewEncapsulation } from "@angular/core";
import { Product } from "./product.model";
import { ProductFormGroup } from "./form.model";
import { Model } from "./repository.model";
import { VALUE_SERVICE } from "./valueDisplay.directive";

@Component({
    selector: "paProductForm",
    templateUrl: "app/productForm.component.html",
    viewProviders: [{ provide: VALUE_SERVICE, useValue: "Oranges" }]
})
export class ProductFormComponent {

    // ...Методы и свойства опущены для краткости...
}
```

Angular использует провайдер при разрешении зависимостей для потомков представления, но не для контентных потомков. Это означает, что зависимости контентных потомков проверяются вверх по иерархии приложения, словно компонент не определил провайдера. В данном примере это означает, что потомок представления получит службу, созданную провайдером компонента, а контентный потомок — службу, созданную провайдером модуля (рис. 20.5).

Рис. 20.5. Определение провайдера для потомков представлений

ВНИМАНИЕ

Определение провайдеров для одной службы с использованием свойств providers и viewProviders не поддерживается. Если вы попробуете это сделать, потомки получат службу, созданную провайдером viewProviders.

Управление разрешением зависимостей

В Angular включены три декоратора, которые могут использоваться для передачи инструкций относительно разрешения зависимостей. Эти декораторы описаны в табл. 20.9, а примеры их использования будут приведены ниже.

Таблица 20.9. Декораторы разрешения зависимостей

Имя	Описание
@Host	Декоратор ограничивает поиск провайдера ближайшим компонентом
@Optional	Декоратор запрещает Angular сообщать об ошибке, если зависимость не удастся разрешить
@SkipSelf	Декоратор исключает провайдеров, определяемых компонентом/директивой, чья зависимость разрешается

Ограничения при поиске провайдера

Декоратор `@Host` ограничивает поиск подходящего провайдера так, чтобы он остановился при достижении ближайшего компонента. Декоратор обычно объединяется с декоратором `@Optional`, который запрещает Angular выдавать исключение, если зависимость службы не удастся разрешить. В листинге 20.29 продемонстрировано применение обоих декораторов к директиве.

Листинг 20.29. Добавление декораторов зависимостей в файле valueDisplay.directive.ts

```
import { Directive, OpaqueToken, Inject,
         HostBinding, Host, Optional} from "@angular/core";

export const VALUE_SERVICE = new OpaqueToken("value_service");

@Directive({
    selector: "[paDisplayValue]"
})
export class PaDisplayValueDirective {

    constructor( @Inject(VALUE_SERVICE) @Host() @Optional() serviceValue: string) {
        this.elementContent = serviceValue || "No Value";
    }

    @HostBinding("textContent")
    elementContent: string;
}
```

При использовании декоратора `@Optional` необходимо позаботиться о том, чтобы класс мог функционировать при невозможности разрешения службы; в этом случае аргумент конструктора службы равен `undefined`. Ближайший компонент определяет службу для своих потомков представления, но не контентных потомков; это означает, что один экземпляр директивы получит объект службы, а другой — нет (рис. 20.6).

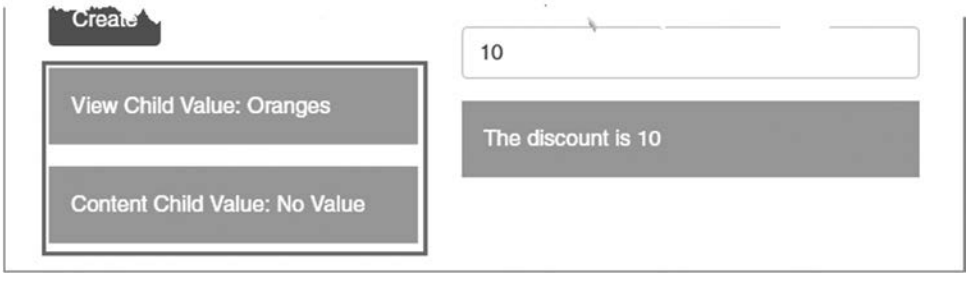


Рис. 20.6. Управление разрешением зависимостей

Игнорирование самоопределяемых провайдеров

По умолчанию провайдеры, определяемые компонентом или директивой, используются для разрешения их зависимостей. Применение декоратора `@SkipSelf` к аргументу конструктора приказывает Angular игнорировать локальных провайдеров и начать поиск на следующем уровне иерархии приложения; это означает, что локальные провайдеры будут использоваться только для разрешения зависимостей потомков. В листинге 20.30 добавляется зависимость для провайдера `VALUE_SERVICE`, помеченного декоратором `@SkipSelf`.

Листинг 20.30. Игнорирование локальных провайдеров в файле productForm.component.ts

```
import { Component, Output, EventEmitter, ViewEncapsulation,
         Inject, SkipSelf } from "@angular/core";
import { Product } from "./product.model";
import { ProductFormGroup } from "./form.model";
import { Model } from "./repository.model";
import { VALUE_SERVICE } from "./valueDisplay.directive";

@Component({
    selector: "paProductForm",
    templateUrl: "app/productForm.component.html",
    viewProviders: [{ provide: VALUE_SERVICE, useValue: "Oranges" }]
})
export class ProductFormComponent {
    form: ProductFormGroup = new ProductFormGroup();
    newProduct: Product = new Product();
    formSubmitted: boolean = false;

    constructor(private model: Model,
```

```
            @Inject(VALUE_SERVICE) @SkipSelf() private serviceValue: string) {
        console.log("Service Value: " + serviceValue);
    }

    submitForm(form: any) {
        this.formSubmitted = true;
        if (form.valid) {
            this.model.saveProduct(this.newProduct);
            this.newProduct = new Product();
            this.form.reset();
            this.formSubmitted = false;
        }
    }
}
```

Когда вы сохраните изменения, а браузер перезагрузит страницу, на консоль JavaScript выводится следующее сообщение, которое показывает, что определенная локально служба была проигнорирована, а зависимость разрешается модулем Angular:

```
Service Value: Apples
```

Итоги

В этой главе объясняется роль провайдеров во внедрении зависимостей и возможности их использования для изменения использования служб в целях разрешения зависимостей. Я описал разные типы провайдеров, которые могут использоваться для создания объектов служб, и продемонстрировал возможности определения директивами и компонентами собственных провайдеров для разрешения их зависимостей и зависимостей их потомков. В следующей главе рассматриваются модули — последняя разновидность структурных блоков приложений Angular.

21 Использование и создание модулей

В этой главе рассматриваются *модули* — последняя разновидность структурных блоков Angular. Первая часть главы посвящена корневому модулю, который используется в любом приложении Angular для описания конфигурации приложения. Во второй части главы рассматриваются *функциональные модули*, используемые для структурирования приложения и группировки взаимосвязанных возможностей. В табл. 21.1 модули представлены в контексте.

Таблица 21.1. Модули в контексте

Вопрос	Ответ
Что это такое?	Модули предоставляют Angular данные конфигурации
Для чего они нужны?	Корневой модуль описывает приложение для Angular и настраивает такие жизненно важные аспекты, как компоненты и службы. Функциональные модули полезны для структурирования сложных проектов, они упрощают операции сопровождения и управления
Как они используются?	Модули представляют собой классы, к которым применен декоратор @NgModule. Свойства, используемые декоратором, имеют разный смысл для корневых и функциональных модулей
Есть ли у них недостатки или скрытые проблемы?	У провайдеров не существует области видимости уровня модуля; это означает, что доступ к провайдерам, определяемым функциональным модулем, будет осуществляться так, как если бы они определялись корневым модулем
Есть ли альтернативы?	Каждое приложение должно содержать корневой модуль, но использование функциональных модулей — дело исключительно добровольное. С другой стороны, без использования функциональных модулей управление приложением быстро усложняется

В табл. 21.2 приведена краткая сводка материала главы.

Таблица 21.2. Сводка материала главы

Проблема	Решение	Листинг
Описание приложения и содержащихся в нем структурных блоков	Используйте корневой модуль	1–7
Группировка взаимосвязанных возможностей	Создайте функциональный модуль	8–28

Подготовка проекта

Как и в других главах этой части книги, мы продолжим работать с проектом, который был создан в главе 11 и расширялся и дорабатывался во всех последующих главах.

ПРИМЕЧАНИЕ

Если вы не хотите создавать проект самостоятельно, исходный код всех глав книги можно загрузить на сайте издательства apress.com.

Чтобы подготовиться к задачам этой главы, удалите часть функциональности из шаблонов компонентов.

В листинге 21.1 приведен шаблон таблицы товаров, в котором были закомментированы элементы компонентов вывода и редактирования скидки.

Листинг 21.1. Содержимое файла productTable.component.html

```
<table class="table table-sm table-bordered table-striped">
    <tr>
        <th></th><th>Name</th><th>Category</th><th>Price</th>
        <th>Discount</th><th></th>
    </tr>
    <tr *paFor="let item of getProducts(); let i = index;
            let odd = odd; let even = even" [class.bg-info]="odd"
            [class.bg-warning]="even">
        <td style="vertical-align:middle">{{i + 1}}</td>
        <td style="vertical-align:middle">{{item.name}}</td>
        <td style="vertical-align:middle">{{item.category}}</td>
        <td style="vertical-align:middle">
            {{item.price | discount | currency:"USD":true }}
        </td>
        <td style="vertical-align:middle" [pa-price]="item.price"
            #discount="discount">
            {{ discount.discountAmount | currency:"USD": true }}
        </td>
        <td class="text-xs-center">
            <button class="btn btn-danger btn-sm" (click)="deleteProduct(item.id)">
                Delete
            </button>
        </td>
    </tr>
</table>
<!--<paDiscountEditor></paDiscountEditor>-->
<!--<paDiscountDisplay></paDiscountDisplay>-->
```

В листинге 21.2 показан шаблон компонента формы товара, в котором были закомментированы элементы, использованные для демонстрации различий между провайдерами потомков представлений и контентных потомков в главе 20.

Листинг 21.2. Содержимое файла productForm.component.html

```
<form novalidate [formGroup]="form" (ngSubmit)="submitForm(form)">
    <div class="form-group" *ngFor="let control of form.productControls">
        <label>{{control.label}}</label>
        <input class="form-control"
            [(ngModel)]="newProduct[control.modelProperty]"
            name="{{control.modelProperty}}"
            formControlName="{{control.modelProperty}}" />
        <ul class="text-danger list-unstyled"
                *ngIf="(formSubmitted || control.dirty) && !control.valid">
            <li *ngFor="let error of control.getValidationMessages()">
                {{error}}
            </li>
        </ul>
    </div>
    <button class="btn btn-primary" type="submit"
        [disabled]="formSubmitted && !form.valid"
        [class.btn-secondary]="formSubmitted && !form.valid">
            Create
    </button>
</form>
<!--<div class="bg-info m-t-1 p-a-1">
    View Child Value: <span paDisplayValue></span>
</div>
<div class="bg-info m-t-1 p-a-1">
    Content Child Value: <ng-content></ng-content>
</div>-->
```

Запустите компилятор TypeScript и сервер HTTP для разработки, выполнив следующую команду из папки example:

```
npm start
```

На экране появляется новое окно (или вкладка) браузера с контентом, показанным на рис. 21.1.

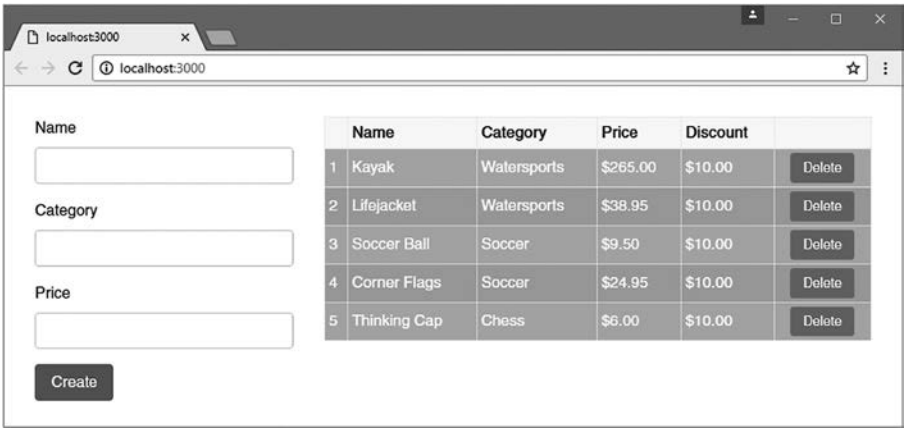


Рис. 21.1. Запуск приложения

Корневой модуль

Каждое приложение Angular содержит по крайней мере один модуль, называемый *корневым модулем*. Корневой модуль традиционно определяется в файле с именем app.module.ts в папке app и содержит класс, к которому был применен декоратор @NgModule. В листинге 21.3 приведен корневой модуль нашего примера.

Листинг 21.3. Корневой модуль в файле app.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { ProductComponent } from "./component";
import { FormsModule, ReactiveFormsModule  } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
import { PaModel } from "./twoway.directive";
import { PaStructureDirective } from "./structure.directive";
import { PaIteratorDirective } from "./iterator.directive";
import { PaCellColor } from "./cellColor.directive";
import { PaCellColorSwitcher } from "./cellColorSwitcher.directive";
import { ProductTableComponent } from "./productTable.component";
import { ProductFormComponent } from "./productForm.component";
import { PaAddTaxPipe } from "./addTax.pipe";
import { PaCategoryFilterPipe } from "./categoryFilter.pipe";
import { PaDiscountDisplayComponent } from "./discountDisplay.component";
import { PaDiscountEditorComponent } from "./discountEditor.component";
import { DiscountService } from "./discount.service";
import { PaDiscountPipe } from "./discount.pipe";
import { PaDiscountAmountDirective } from "./discountAmount.directive";
import { SimpleDataSource } from "./datasource.model";
import { Model } from "./repository.model";
import { LogService, LOG_SERVICE, SpecialLogService,
    LogLevel, LOG_LEVEL} from "./log.service";
import { VALUE_SERVICE, PaDisplayValueDirective} from "./valueDisplay.directive";

@NgModule({
    imports: [BrowserModule, FormsModule, ReactiveFormsModule],
    declarations: [ProductComponent, PaAttrDirective, PaModel,
        PaStructureDirective, PaIteratorDirective,
        PaCellColor, PaCellColorSwitcher, ProductTableComponent,
        ProductFormComponent, PaAddTaxPipe, PaCategoryFilterPipe,
        PaDiscountDisplayComponent, PaDiscountEditorComponent,
        PaDiscountPipe, PaDiscountAmountDirective, PaDisplayValueDirective],
    providers: [DiscountService, SimpleDataSource, Model, LogService,
        { provide: VALUE_SERVICE, useValue: "Apples" }],
    bootstrap: [ProductComponent]
})
export class AppModule { }
```

В проекте может быть несколько модулей, но именно корневой модуль используется в файле начальной загрузки, который традиционно называется main.ts

и определяется в папке app. В листинге 21.4 представлен файл main.ts из нашего примера.

МОДУЛИ ANGULAR И МОДУЛИ JAVASCRIPT

В приложениях Angular встречаются модули двух типов: модули Angular и модули JavaScript. Модуль Angular представляет собой класс, к которому был применен декоратор @NgModule. В каждом приложении имеется корневой модуль Angular, а новые модули Angular добавляются в приложение через свойство imports корневого компонента. В листинге 21.3 модуль FormsModule включается в свойство imports корневого модуля, открывая доступ к содержащейся в нем информации:

```
...
imports: [BrowserModule, FormsModule, ReactiveFormsModule],
...
```

Браузер должен загрузить код модуля Angular; здесь в игру вступает пакет JavaScript. Для введения типа FormsModule в область видимости в файл app.module.ts включается команда import следующего вида:

```
...
import { FormsModule, ReactiveFormsModule  } from "@angular/forms";
...
```

Ссылка the @angular/forms относится к модулю Javascript. Загрузчик модулей Javascript (пакет SystemJS в нашем примере) связывает ссылку @angular/forms с файлом node_modules/@angular/forms/bundles/forms.umd.min.js — сокращенным файлом с кодом Javascript, который содержит несколько модулей Angular.

Таким образом, хотя термин «модуль» в приложениях Angular имеет два разных значения, эти значения связаны друг с другом. Модули Angular — классы, предоставляющие доступ к функциональности Angular и упакованные для передачи браузеру в модулях JavaScript.

Листинг 21.4. Файл начальной загрузки Angular main.ts

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

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

Приложения Angular могут работать в разных средах, таких как веб-браузеры и платформенные контейнеры приложений. Задача файла начальной загрузки — выбор платформы и идентификация корневого модуля. Метод `platformBrowserDynamic` создает исполнительную среду браузера, а метод `bootstrapModule` назначает модуль, которым является класс `AppModule` из листинга 21.3.

При определении корневого модуля используются свойства декоратора `@NgModule`, описанные в табл. 21.3. (Также существуют другие свойства декоратора, описанные позже в этой главе.)

Таблица 21.3. Свойства декоратора корневого модуля @NgModule

Имя	Описание
imports	Свойство задает модули Angular, необходимые для поддержки директив, компонентов и каналов в приложении
declarations	Свойство задает директивы, компоненты и каналы, используемые в приложении
providers	Свойство задает провайдеров служб, которые будут использоваться инжектором модуля. Эти провайдеры, доступные во всем приложении, будут использоваться при отсутствии локального провайдера для службы (см. главу 20)
bootstrap	Свойство задает корневой компонент приложения

Свойство imports

Свойство `imports` используется для перечисления других модулей, необходимых приложению. В нашем примере это все модули, предоставляемые Angular:

```
...
imports: [BrowserModule, FormsModule, ReactiveFormsModule],
...
```

Модуль `BrowserModule` предоставляет функциональность, необходимую для запуска приложений Angular в браузерах. Два других модуля предоставляют поддержку для работы с формами HTML и формами на базе моделей (см. главу 14). Также существуют другие модули Angular, описанные в последующих главах.

Свойство `imports` также используется для объявления зависимостей нестандартных модулей в целях управления сложными приложениями Angular и создания блоков функциональности, предназначенной для повторного использования. О том, как определяются нестандартные модули, я расскажу в разделе «Создание функциональных модулей».

Свойство declarations

Свойство `declarations` передает Angular список директив, компонентов и каналов, необходимых приложению (они объединяются под общим термином «*объявляемые классы*» (declarable classed). Свойство `declarations` в корневом модуле примера содержит длинный список классов, каждый из которых может использоваться в любой точке приложения — просто потому, что он указан в этом свойстве.

```
...
declarations: [ProductComponent, PaAttrDirective, PaModel,
    PaStructureDirective, PaIteratorDirective,
    PaCellColor, PaCellColorSwitcher, ProductTableComponent,
    ProductFormComponent, PaAddTaxPipe, PaCategoryFilterPipe,
    PaDiscountDisplayComponent, PaDiscountEditorComponent,
    PaDiscountPipe, PaDiscountAmountDirective, PaDisplayValueDirective],
...
```

Обратите внимание: встроенные объявляемые классы (такие, как директивы, описанные в главе 13, и каналы, описанные в главе 18) не включаются в свойство `declarations` корневого модуля. Дело в том, что ими управляет `BrowserModule` и при добавлении этого модуля в свойство `imports` его объявляемые классы автоматически становятся доступными для использования в приложении.

Свойство providers

Свойство `providers` определяет провайдеров служб, которые будут использоваться для разрешения зависимостей при отсутствии подходящих локальных провайдеров. Использование провайдеров служб подробно описано в главах 19 и 20.

Свойство bootstrap

Свойство `bootstrap` задает корневой компонент или компоненты приложения. В процессе обработки основного документа HTML, который традиционно называется index.html, Angular анализирует корневые компоненты и применяет их с использованием значения свойства `selector` в декораторах `@Component`.

ПРИМЕЧАНИЕ

Компоненты, перечисленные в свойстве bootstrap, также должны быть включены в список declarations.

Свойство `bootstrap` из корневого модуля примера выглядит так:

```
...
bootstrap: [ProductComponent]
...
```

Класс `ProductComponent` предоставляет корневой компонент, а свойство `selector` задает элемент `app` (листинг 21.5).

Листинг 21.5. Корневой компонент в файле component.ts

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

@Component({
    selector: "app",
    templateUrl: "app/template.html"
})
export class ProductComponent {

}
```

Когда мы только начинали работу над проектом в главе 11, корневой компонент содержал большой объем функциональности. Но с введением дополнительных компонентов роль компонента сократилась, и теперь он по сути представляет собой заполнитель, который приказывает Angular спроецировать содержимое фай-

ла app/template.html в элемент app документа HTML; таким образом загружаются компоненты, выполняющие настоящую работу в приложении.

В таком подходе нет ничего плохого, но он означает, что на долю корневого компонента в приложении остается не так уж много работы. Если подобная избыточность покажется вам неэлегантной, вы можете указать несколько корневых компонентов в корневом модуле, и все они будут использоваться для элементов в документе HTML. Для демонстрации я удалил существующий корневой компонент из свойства bootstrap корневого модуля и заменил его классами компонентов, ответственными за форму товара и таблицу товаров (листинг 21.6).

Листинг 21.6. Указание множественных корневых компонентов в файле app.module.ts

```
...
@NgModule({
    imports: [BrowserModule, FormsModule, ReactiveFormsModule],
    declarations: [ProductComponent, PaAttrDirective, PaModel,
        PaStructureDirective, PaIteratorDirective,
        PaCellColor, PaCellColorSwitcher, ProductTableComponent,
        ProductFormComponent, PaAddTaxPipe, PaCategoryFilterPipe,
        PaDiscountDisplayComponent, PaDiscountEditorComponent,
        PaDiscountPipe, PaDiscountAmountDirective, PaDisplayValueDirective],
    providers: [DiscountService, SimpleDataSource, Model, LogService,
        { provide: VALUE_SERVICE, useValue: "Apples" }],
    bootstrap: [ProductFormComponent, ProductTableComponent]
})
export class AppModule { }
...
```

В листинге 21.7 отражены изменения в корневых компонентах главного документа HTML.

Листинг 21.7. Изменение элементов корневых компонентов в файле index.html

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
    <meta charset="utf-8" />
    <script src="node_modules/classlist.js/classList.min.js"></script>
    <script src="node_modules/core-js/client/shim.min.js"></script>
    <script src="node_modules/intl/dist/Intl.complete.js"></script>
    <script src="node_modules/zone.js/dist/zone.min.js"></script>
    <script src="node_modules/reflect-metadata/Reflect.js"></script>
    <script src="node_modules/systemjs/dist/system.src.js"></script>
    <script src="systemjs.config.js"></script>
    <script>
      System.import("app/main").catch(function(err){ console.error(err); });
    </script>
    <link href="node_modules/bootstrap/dist/css/bootstrap.min.css"
        rel="stylesheet" />
</head>
<body class="m-a-1">
    <div class="col-xs-8 p-a-1">
```

```
        <paProductTable></paProductTable>
    </div>
    <div class="col-xs-4 p-a-1">
        <paProductForm></paProductForm>
    </div>
</body>
</html>
```

Я изменил порядок следования этих компонентов по сравнению с предыдущими примерами просто для того, чтобы создать видимое изменение в макете приложения. Когда все изменения будут сохранены, а браузер перезагрузит страницу, вы увидите новые корневые компоненты (рис. 21.2).

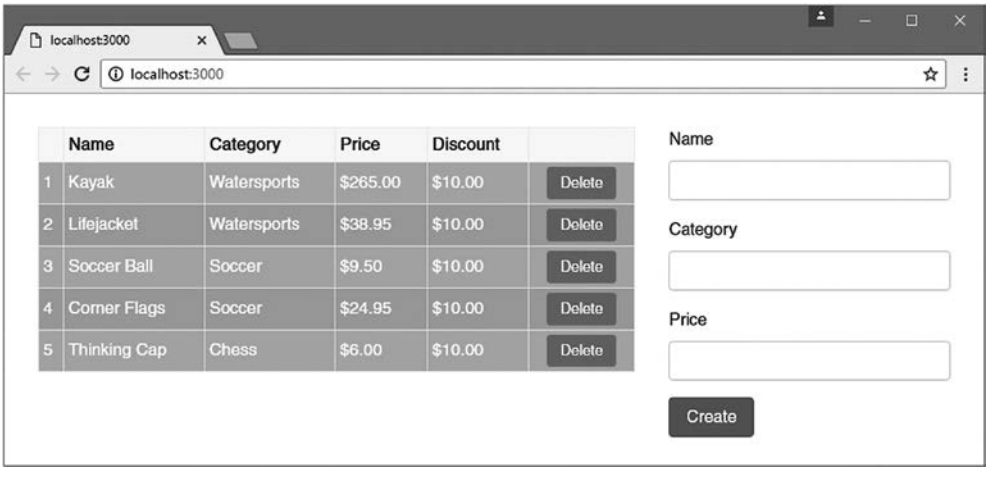


Рис. 21.2. Использование множественных корневых компонентов

Провайдеры служб модуля используются для разрешения зависимостей всех корневых компонентов. В нашем примере это означает, что существует один объект службы `Model`, который совместно используется в приложении и позволяет автоматически выводить в таблице товары, созданные на форме HTML, — при том, что эти компоненты были преобразованы в корневые компоненты.

Создание функциональных модулей

По мере описания различных функциональных возможностей нашего приложения корневой модуль непрерывно усложнялся: он содержит длинный список команд `import` для загрузки модулей JavaScript и набор классов в свойстве `declarations` декоратора `@NgModule`, который занимает несколько строк (листинг 21.8).

Листинг 21.8. Содержимое файла app.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { ProductComponent } from "./component";
```

```
import { FormsModule, ReactiveFormsModule  } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
import { PaModel } from "./twoway.directive";
import { PaStructureDirective } from "./structure.directive";
import { PaIteratorDirective } from "./iterator.directive";
import { PaCellColor } from "./cellColor.directive";
import { PaCellColorSwitcher } from "./cellColorSwitcher.directive";
import { ProductTableComponent } from "./productTable.component";
import { ProductFormComponent } from "./productForm.component";
import { PaAddTaxPipe } from "./addTax.pipe";
import { PaCategoryFilterPipe } from "./categoryFilter.pipe";
import { PaDiscountDisplayComponent } from "./discountDisplay.component";
import { PaDiscountEditorComponent } from "./discountEditor.component";
import { DiscountService } from "./discount.service";
import { PaDiscountPipe } from "./discount.pipe";
import { PaDiscountAmountDirective } from "./discountAmount.directive";
import { SimpleDataSource } from "./datasource.model";
import { Model } from "./repository.model";
import { LogService, LOG_SERVICE, SpecialLogService,
    LogLevel, LOG_LEVEL} from "./log.service";
import { VALUE_SERVICE, PaDisplayValueDirective} from "./valueDisplay.directive";

@NgModule({
    imports: [BrowserModule, FormsModule, ReactiveFormsModule],
    declarations: [ProductComponent, PaAttrDirective, PaModel,
        PaStructureDirective, PaIteratorDirective,
        PaCellColor, PaCellColorSwitcher, ProductTableComponent,
        ProductFormComponent, PaAddTaxPipe, PaCategoryFilterPipe,
        PaDiscountDisplayComponent, PaDiscountEditorComponent,
        PaDiscountPipe, PaDiscountAmountDirective, PaDisplayValueDirective],
    providers: [DiscountService, SimpleDataSource, Model, LogService,
        { provide: VALUE_SERVICE, useValue: "Apples" }],
    bootstrap: [ProductFormComponent, ProductTableComponent]
})
export class AppModule { }
```

Функциональные модули используются для группировки взаимосвязанной функциональности, чтобы ее можно было использовать как единое целое (по аналогии с модулями Angular). Например, когда потребуется функциональность для работы с формами, мне не нужно будет добавлять команды `import` и записи `declarations` для каждой отдельной директивы, компонента или канала. Вместо этого достаточно добавить `BrowserModule` в свойство `imports` декоратора, и вся содержащаяся в нем функциональность становится доступной в приложении.

При создании функционального модуля вы можете либо сосредоточиться на некоторой функциональной области приложения, либо сгруппировать набор взаимосвязанных структурных блоков, формирующих инфраструктуру приложения. В дальнейших разделах я продемонстрирую оба варианта, потому что они работают немного по-разному и требуют учета разных факторов. Функциональные модули используют один декоратор `@NgModule`, но с другим набором свойств конфигурации — одни новые, другие уже знакомы по корневому модулю, но работают

иначе. О том, как используются эти свойства, я расскажу ниже, а в табл. 21.4 приведена краткая сводка.

Таблица 21.4. Свойства декоратора @NgModule для функциональных модулей

Имя	Описание
imports	Свойство используется для импортирования модулей, необходимых классам в модулях
providers	Свойство задает провайдеров для модуля. При загрузке функционального модуля его набор провайдеров объединяется с провайдерами корневого модуля; это означает, что службы функционального модуля доступны в приложении (а не только внутри модуля)
declarations	Свойство задает директивы, компоненты и каналы для модуля. Свойство должно содержать как классы, используемые внутри модуля, так и классы, доступные во всем приложении
exports	Свойство используется для определения открытых экспортируемых аспектов модуля. Оно содержит (частично или полностью) директивы, компоненты и каналы из свойства declarations, а также модули из свойства imports

Создание модуля модели

Термин «*модуль модели*» больше смахивает на скороговорку, но обычно он становится хорошей отправной точкой для рефакторинга приложения с использованием функциональных модулей — просто потому, что все остальные структурные блоки в приложении зависят от модели.

Все начинается с создания папки, в которой будет содержаться модуль. Папки модулей создаются в папке app, и им присваиваются содержательные имена. Для этого модуля создается папка app/model.

Схемы назначения имен, используемые для файлов Angular, упрощают перемещение и удаление групп файлов. Выполните следующие команды из папки example, чтобы переместить файлы (команды работают в Windows PowerShell, Linux и macOS):

```
mv app/*.model.ts app/model/
mv app/limit.formvalidator.ts app/model/
```

После того как все файлы будут перемещены, удалите соответствующие файлы JavaScript, выполнив следующие команды из папки example:

```
rm app/*.model.js
rm app/limit.formvalidator.js
```

В результате файлы, перечисленные в табл. 21.5, перемещаются в папку model.

Таблица 21.5. Перемещение файлов для модуля

Файл	Новое местоположение
app/datasource.model.ts	app/model/datasource.model.ts
app/form.model.ts	app/model/form.model.ts
app/limit.formvalidator.ts	app/model/limit.formvalidator.ts
app/product.model.ts	app/model/product.model.ts
app/repository.model.ts	app/model/repository.model.ts

Когда перемещение и удаление файлов будет завершено, компилятор TypeScript выдаст серию ошибок компилятора, потому что некоторые ключевые объявляемые классы стали недоступными, а браузер сообщает об ошибках, потому что он не может найти удаленные файлы JavaScript. О том, как справиться с этими проблемами, будет рассказано в следующих разделах.

Создание определения модуля

На следующем шаге определяется модуль, который сводит воедино всю функциональность в файлах, перемещенных в новую папку. Создайте файл model.module.ts в папке app/model и включите в него определение модуля из листинга 21.9.

Листинг 21.9. Содержимое файла model.module.ts в папке app/model

```
import { NgModule } from "@angular/core";
import { SimpleDataSource } from "./datasource.model";
import { Model } from "./repository.model";

@NgModule({
    providers: [Model, SimpleDataSource]
})
export class ModelModule { }
```

Даже этот простой модуль обладает некоторыми важными характеристиками, которые необходимо понять, потому что цель функционального модуля — избирательное предоставление доступа к содержимому папки модуля в коде приложения. Декоратор `@NgModule` этого модуля использует только свойство `providers` для определения провайдеров классов для служб `Model` и `SimpleDataSource`. Провайдеры, используемые в функциональном модуле, регистрируются с инжектором корневого модуля; это означает, что они будут доступны во всем приложении — именно то, что нужно для модели данных нашего примера.

ПРИМЕЧАНИЕ

Многие разработчики ошибочно полагают, что службы, определенные в модуле, доступны только для классов внутри этого модуля. В Angular не существует области видимости уровня модуля. Провайдеры, определенные функциональным модулем, используются так, словно они определяются корневым модулем. Локальные провайдеры, определяемые директивами и компонентами функционального модуля, доступны для их потомков представлений и контентных потомков, даже если они определяются в других модулях.

Обновление других классов в приложении

Перемещение классов в папку model нарушило работу команд `import` в других частях приложения, зависящих от класса `Product` или `ProductFormGroup`. На следующем шаге эти команды `import` следует обновить ссылками на новый модуль. Изменения затрагивают четыре файла: attr.directive.ts, categoryFilter.pipe.ts, productForm.component.ts и productTable.component.ts. В листинге 21.10 показаны необходимые изменения в файле attr.directive.ts.

Листинг 21.10. Обновление ссылки import в файле attr.directive.ts

```
import { Directive, ElementRef, Attribute, Input,
    SimpleChange, Output, EventEmitter, HostListener, HostBinding }
from "@angular/core";
import { Product } from "./model/product.model";

@Directive({
    selector: "[pa-attr]"
})
export class PaAttrDirective {

    // ...Команды опущены для краткости...
}
```

Единственное необходимое изменение — обновление пути в команде `import` в соответствии с новым местоположением файла с программным кодом. В листинге 21.11 показаны те же изменения, примененные в файле categoryFilter.pipe.ts.

Листинг 21.11. Обновление ссылки import в файле categoryFilter.pipe.ts

```
import { Pipe } from "@angular/core";
import { Product } from "./model/product.model";

@Pipe({
    name: "filter",
    pure: false
})
export class PaCategoryFilterPipe {

    transform(products: Product[], category: string): Product[] {
        return category == undefined ?
            products : products.filter(p => p.category == category);
    }
}
```

В листинге 21.12 обновляются команды `import` в файле productForm.component.ts.

Листинг 21.12. Обновление пути import в файле productForm.component.ts

```
import { Component, Output, EventEmitter, ViewEncapsulation,
    Inject, SkipSelf } from "@angular/core";
import { Product } from "./model/product.model";
import { ProductFormGroup } from "./model/form.model";
import { Model } from "./model/repository.model";
```

```
import { VALUE_SERVICE } from "./valueDisplay.directive";

@Component({
    selector: "paProductForm",
    templateUrl: "app/productForm.component.html",
    viewProviders: [{ provide: VALUE_SERVICE, useValue: "Oranges" }]
})
export class ProductFormComponent {

    // ...Команды опущены для краткости...
}
```

В листинге 21.13 обновляются пути в последнем файле, productTable.component.ts.

Листинг 21.13. Обновление пути import в файле productForm.component.ts

```
import { Component, Input, ViewChildren, QueryList } from "@angular/core";
import { Model } from "./model/repository.model";
import { Product } from "./model/product.model";
import { DiscountService } from "./discount.service";

@Component({
    selector: "paProductTable",
    templateUrl: "app/productTable.component.html"
})
export class ProductTableComponent {

    // ...Команды опущены для краткости...
}
```

Обновление корневого модуля

Осталось сделать последний шаг — обновить корневой модуль, чтобы службы, определенные в функциональном модуле, были доступны во всем приложении. В листинге 21.14 показаны необходимые изменения.

Листинг 21.14. Обновление корневого модуля в файле app.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { ProductComponent } from "./component";
import { FormsModule, ReactiveFormsModule } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
import { PaModel } from "./twoway.directive";
import { PaStructureDirective } from "./structure.directive";
import { PaIteratorDirective } from "./iterator.directive";
import { PaCellColor } from "./cellColor.directive";
import { PaCellColorSwitcher } from "./cellColorSwitcher.directive";
import { ProductTableComponent } from "./productTable.component";
import { ProductFormComponent } from "./productForm.component";
import { PaAddTaxPipe } from "./addTax.pipe";
```

```
import { PaCategoryFilterPipe } from "./categoryFilter.pipe";
import { PaDiscountDisplayComponent } from "./discountDisplay.component";
import { PaDiscountEditorComponent } from "./discountEditor.component";
import { DiscountService } from "./discount.service";
import { PaDiscountPipe } from "./discount.pipe";
import { PaDiscountAmountDirective } from "./discountAmount.directive";
import { ModelModule } from "./model/model.module";
import {
    LogService, LOG_SERVICE, SpecialLogService,
    LogLevel, LOG_LEVEL
} from "./log.service";
import { VALUE_SERVICE, PaDisplayValueDirective } from "./valueDisplay.directive";

@NgModule({
    imports: [BrowserModule, FormsModule, ReactiveFormsModule, ModelModule],
    declarations: [ProductComponent, PaAttrDirective, PaModel,
        PaStructureDirective, PaIteratorDirective,
        PaCellColor, PaCellColorSwitcher, ProductTableComponent,
        ProductFormComponent, PaAddTaxPipe, PaCategoryFilterPipe,
        PaDiscountDisplayComponent, PaDiscountEditorComponent,
        PaDiscountPipe, PaDiscountAmountDirective, PaDisplayValueDirective],
    providers: [DiscountService, LogService,
        { provide: VALUE_SERVICE, useValue: "Apples" }],
    bootstrap: [ProductFormComponent, ProductTableComponent]
})
export class AppModule { }
```

Я импортировал функциональный модуль и добавил его в список imports корневого модуля. Так как функциональный модуль определяет провайдеров для `Model` и `SimpleDataSource`, я удалил записи из списка `providers` корневого модуля и соответствующие команды `import`.

При сохранении изменений в корневом модуле браузер перезагружает приложение. Корневой модуль загружает функциональный модуль модели, который предоставляет доступ к службам модели, и приложение запускается правильно.

Создание вспомогательного функционального модуля

Модуль модели стал хорошей отправной точкой, потому что он демонстрирует базовую структуру функционального модуля и его связь с корневым модулем. Впрочем, на приложение он повлиял не сильно, и особого упрощения не достигнуто.

Следующим шагом станет создание вспомогательного функционального модуля, который группирует всю общую функциональность приложения (например, каналы и директивы). В реальном проекте к группировке структурных элементов такого рода следует относиться более тщательно; обычно в приложении создаются несколько модулей, группирующих сходную функциональность. В нашем примере все каналы, директивы и службы вынесены в один модуль.

Создание папки модуля и перемещение файлов

Как и с предыдущим модулем, все начинается с создания папки. Для этого модуля создайте папку app/common. Выполните следующую команду из папки example, чтобы переместить файлы TypeScript для каналов и директив:

```
mv app/*.pipe.ts app/common/
mv app/*.directive.ts app/common/
```

Команды работают в Windows PowerShell, Linux и macOS. Некоторые из директив и каналов приложения зависят от классов `DiscountService` и `LogServices`, которые предоставляются им через механизм внедрения зависимостей. Выполните следующую команду из папки app, чтобы переместить файл TypeScript для службы в папку module:

```
mv app/*.service.ts app/common/
```

После перемещения файлов TypeScript удалите соответствующие файлы JavaScript, выполнив следующие команды из папки example:

```
rm app/*.pipe.js
rm app/*.directive.js
rm app/*.service.js
```

В результате файлы, перечисленные в табл. 21.6, перемещаются в папку модуля.

Таблица 21.6. Перемещение файлов для модуля

Файл	Новое местоположение
app/addTax.pipe.ts	app/common/addTax.pipe.ts
app/attr.directive.ts	app/common/attr.directive.ts
app/categoryFilter.pipe.ts	app/common/categoryFilter.pipe.ts
app/cellColor.directive.ts	app/common/cellColor.directive.ts
app/cellColorSwitcher.directive.ts	app/common/cellColorSwitcher.directive.ts
app/discount.pipe.ts	app/common/discount.pipe.ts
app/discountAmount.directive.ts	app/common/discountAmount.directive.ts
app/iterator.directive.ts	app/common/iterator.directive.ts
app/structure.directive.ts	app/common/structure.directive.ts
app/twoway.directive.ts	app/common/twoway.directive.ts
app/valueDisplay.directive.ts	app/common/valueDisplay.directive.ts
app/discount.service.ts	app/common/discount.service.ts
app/log.service.ts	app/common/log.service.ts

Обновление классов в новом модуле

В некоторых классах, перемещенных в новую папку, содержатся команды `import`, которые необходимо обновить в соответствии с новым путем к модулю модели. В листинге 21.15 показаны изменения, которые необходимо внести в файл attr.directive.ts.

Листинг 21.15. Обновление команды import в файле attr.directive.ts

```
import {
    Directive, ElementRef, Attribute, Input,
    SimpleChange, Output, EventEmitter, HostListener, HostBinding
}
    from "@angular/core";
import { Product } from "../model/product.model";

@Directive({
    selector: "[pa-attr]"
})

export class PaAttrDirective {
    // ...Команды опущены для краткости...
}
```

В листинге 21.16 показаны соответствующие изменения в файле categoryFilter.pipe.ts.

Листинг 21.16. Обновление команды import в файле categoryFilter.pipe.ts

```
import { Pipe } from "@angular/core";
import { Product } from "../model/product.model";

@Pipe({
    name: "filter",
    pure: false
})
export class PaCategoryFilterPipe {

    transform(products: Product[], category: string): Product[] {
        return category == undefined ?
            products : products.filter(p => p.category == category);
    }
}
```

Создание определения модуля

Следующий шаг — определение модуля, который объединяет функциональность файлов, перемещенных в новую папку. Создайте файл common.module.ts в папке app/common и включите в него определение модуля из листинга 21.17.

Листинг 21.17. Содержимое файла common.module.ts в папке app/common

```
import { NgModule } from "@angular/core";
import { PaAddTaxPipe } from "./addTax.pipe";
import { PaAttrDirective } from "./attr.directive";
import { PaCategoryFilterPipe } from "./categoryFilter.pipe";
import { PaCellColor } from "./cellColor.directive";
import { PaCellColorSwitcher } from "./cellColorSwitcher.directive";
import { PaDiscountPipe } from "./discount.pipe";
import { PaDiscountAmountDirective } from "./discountAmount.directive";
import { PaIteratorDirective } from "./iterator.directive";
import { PaStructureDirective } from "./structure.directive";
import { PaModel } from "./twoway.directive";
import { VALUE_SERVICE, PaDisplayValueDirective} from "./valueDisplay.directive";
import { DiscountService } from "./discount.service";
import { LogService } from "./log.service";
import { ModelModule } from "../model/model.module";

@NgModule({
    imports: [ModelModule],
    providers: [LogService, DiscountService,
        { provide: VALUE_SERVICE, useValue: "Apples" }],
    declarations: [PaAddTaxPipe, PaAttrDirective, PaCategoryFilterPipe,
        PaCellColor, PaCellColorSwitcher, PaDiscountPipe,
        PaDiscountAmountDirective, PaIteratorDirective, PaStructureDirective,
        PaModel, PaDisplayValueDirective],
    exports: [PaAddTaxPipe, PaAttrDirective, PaCategoryFilterPipe,
        PaCellColor, PaCellColorSwitcher, PaDiscountPipe,
        PaDiscountAmountDirective, PaIteratorDirective, PaStructureDirective,
        PaModel, PaDisplayValueDirective]
})
export class CommonModule { }
```

Этот модуль сложнее того, который требовался для модели данных. Ниже рассматриваются значения, использованные для каждого из свойств декоратора.

Свойство imports

Некоторые директивы и каналы модуля зависят от служб, определенных в модуле модели, созданном ранее в этой главе. Чтобы функциональность из этого модуля была доступна в приложении, я добавил его в свойство `imports` модуля common.

Свойство providers

Свойство `providers` обеспечивает доступ директив и каналов из функционального модуля к необходимым им службам. Для этого включаются провайдеры классов для создания служб `LogService` и `DiscountService`, которые будут добавлены к провайдерам корневого модуля при загрузке модуля.

Службы будут доступны не только для директив и каналов модуля common; они становятся доступными во всем приложении.

Свойство declarations

Свойство передает Angular список директив и каналов (и компонентов, если они есть) в модуле. В функциональном модуле это свойство имеет двойное назначение: оно включает возможность использования объявляемых классов в любых шаблонах, содержащихся в модуле, и позволяет модулю открыть доступ к этим объявляемым классам за своими пределами. Позже в этой главе мы создадим модуль, содержащий шаблонный контент, но для этого модуля свойство `declarations` служит другой цели: оно необходимо для использования свойства `exports`, описанного в следующем разделе.

Свойство exports

Для модуля, содержащего директивы и каналы, предназначенные для использования в других местах приложения, свойство `exports` становится самым важным свойством декоратора `@NgModule`, потому что оно определяет набор директив, компонентов и каналов, предоставляемых модулем при его импортировании другими частями приложения. Свойство `exports` может содержать отдельные классы и типы модулей, хотя и те и другие могут быть уже включены в свойство `declarations` или `imports`. При импортировании модуля перечисленные типы ведут себя так, как если бы они были добавлены в свойство `declarations` импортирующего модуля.

Обновление других классов в приложении

Теперь, когда модуль был определен, можно обновить другие файлы в приложении, которые содержат команды `import` для типов, ставших частью модуля common. В листинге 21.18 показаны необходимые изменения в файле discountDisplay.component.ts.

Листинг 21.18. Обновление ссылки import в файле discountDisplay.component.ts

```
import { Component, Input } from "@angular/core";
import { DiscountService } from "./common/discount.service";

@Component({
    selector: "paDiscountDisplay",
    template: `<div class="bg-info p-a-1">
                The discount is {{discounter.discount}}
               </div>`
})
export class PaDiscountDisplayComponent {

    constructor(private discounter: DiscountService) { }
}
```

В листинге 21.19 показаны изменения в файле discountEditor.component.ts.

Листинг 21.19. Обновление ссылки import в файле discountEditor.component.ts

```
import { Component, Input } from "@angular/core";
import { DiscountService } from "./common/discount.service";

@Component({
    selector: "paDiscountEditor",
    template: `<div class="form-group">
                    <label>Discount</label>
                    <input [(ngModel)]="discounter.discount"
                         class="form-control" type="number" />
                </div>`
})
export class PaDiscountEditorComponent {

    constructor(private discounter: DiscountService) { }
}
```

В листинге 21.20 показаны изменения в файле productForm.component.ts.

Листинг 21.20. Обновление ссылки import в файле productForm.component.ts

```
import { Component, Output, EventEmitter, ViewEncapsulation,
    Inject, SkipSelf } from "@angular/core";
import { Product } from "./model/product.model";
import { ProductFormGroup } from "./model/form.model";
import { Model } from "./model/repository.model";
import { VALUE_SERVICE } from "./common/valueDisplay.directive";

@Component({
    selector: "paProductForm",
    templateUrl: "app/productForm.component.html",
    viewProviders: [{ provide: VALUE_SERVICE, useValue: "Oranges" }]
})
export class ProductFormComponent {

    // ...Команды опущены для краткости...
}
```

Последнее изменение в файле productTable.component.ts показано в листинге 21.21.

Листинг 21.21. Обновление ссылки import в файле productTable.component.ts

```
import { Component, Input, ViewChildren, QueryList } from "@angular/core";
import { Model } from "./model/repository.model";
import { Product } from "./model/product.model";
import { DiscountService } from "./common/discount.service";

@Component({
    selector: "paProductTable",
    templateUrl: "app/productTable.component.html"
})
export class ProductTableComponent {

    // ...Команды опущены для краткости...
}
```

Обновление корневого модуля

Остается сделать последний шаг — обновить корневой модуль, чтобы он загружал модуль common для предоставления доступа к содержащимся в нем директивам и каналам (листинг 21.22).

Листинг 21.22. Импортирование функционального модуля в файле app.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { ProductComponent } from "./component";
import { FormsModule, ReactiveFormsModule } from "@angular/forms";
import { ProductTableComponent } from "./productTable.component";
import { ProductFormComponent } from "./productForm.component";
import { PaDiscountDisplayComponent } from "./discountDisplay.component";
import { PaDiscountEditorComponent } from "./discountEditor.component";
import { ModelModule } from "./model/model.module";
import { CommonModule } from "./common/common.module";

@NgModule({
    imports: [BrowserModule, FormsModule, ReactiveFormsModule,
        ModelModule, CommonModule],
    declarations: [ProductComponent, ProductTableComponent,
        ProductFormComponent, PaDiscountDisplayComponent,
PaDiscountEditorComponent],
    bootstrap: [ProductFormComponent, ProductTableComponent]
})
export class AppModule { }
```

Корневой модуль был существенно упрощен созданием модуля common, который был добавлен в список `imports`. Все индивидуальные классы директив и каналов были исключены из списка `declarations`, а соответствующие команды `import` были удалены из файла. При импортировании модуля common все типы, перечисленные в свойстве `exports`, будут добавлены в свойство `declarations` корневого модуля.

Создание функционального модуля с компонентами

Последний модуль, который мы создадим, будет содержать компоненты приложения, чтобы я мог продемонстрировать работу с внешними шаблонами. Базовый процесс создания модуля не отличается от описанного выше.

Создание папки модуля и перемещение файлов

Модулю будет присвоено имя `components`, поэтому для хранения файлов создается папка app/components. Выполните следующую команду из папки example, чтобы переместить файлы TypeScript, HTML и CSS в новую папку и удалить соответствующие файлы JavaScript:

```
mv app/*.component.ts app/components/
mv app/*.component.html app/components/
mv app/*.component.css app/components/
rm app/*.component.js
```

В результате выполнения всех этих команд файлы с кодом директивы перемещаются в новую папку, как показано в табл. 21.7.

Таблица 21.7. Перемещаемые файлы для модуля компонента

Файл	Новое местоположение
app/discountDisplay.component.ts	app/component/discountDisplay.component.ts
app/discountEditor.component.ts	app/component/discountEditor.component.ts
app/productForm.component.ts	app/component/productForm.component.ts
app/productTable.component.ts	app/component/productTable.component.ts

Обновление URL шаблонов

Два компонента из нового модуля используют внешние шаблоны, местоположение которых задается при помощи свойства `templateUrl` декораторов `@Component`. Теперь, после перемещения файлов, их URL-адреса стали недействительными, потому что они указывают на старое местоположение файлов HTML. Проблему можно решить двумя способами. Более простое решение — обновить значение свойства `templateUrl`, чтобы оно соответствовало новому местоположению файла HTML (листинг 21.23). Оно также обновляет пути к другим модулям в командах `import`.

Листинг 21.23. Обновление местоположения шаблона в файле productTable.component.ts

```
import { Component, Input } from "@angular/core";
import { Model } from "../model/repository.model";
import { Product } from "../model/product.model";
import { DiscountService } from "../common/discount.service";

@Component({
    selector: "paProductTable",
    templateUrl: "app/components/productTable.component.html"
})
export class ProductTableComponent {

    // ...Команды опущены для краткости...
}
```

Решение легко реализуется, но оно означает, что вам придется изменять URL шаблона при рефакторинге приложения с разбиением на модули. Другое, более гибкое решение — воспользоваться свойством `moduleId` и функциональностью загрузчика модулей JavaScript для автоматической настройки URL (листинг 21.24).

Листинг 21.24. Настройка идентификатора модуля в файле productForm.component.ts

```
import {
    Component, Output, EventEmitter, ViewEncapsulation,
    Inject, SkipSelf
} from "@angular/core";
import { Product } from "../model/product.model";
import { ProductFormGroup } from "../model/form.model";
import { Model } from "../model/repository.model";
import { VALUE_SERVICE } from "../common/valueDisplay.directive";

@Component({
    selector: "paProductForm",
    moduleId: module.id,
    templateUrl: "productForm.component.html",
    viewProviders: [{ provide: VALUE_SERVICE, useValue: "Oranges" }]
})
export class ProductFormComponent {

    // ...Команды опущены для краткости...
}
```

Свойство `module.id` предоставляется на стадии выполнения; оно содержит модуль с классом компонента. Присваивая это значение свойству `moduleId`, вы приказываете Angular запросить значение `templateUrl` относительно местоположения модуля, которое уже не нужно жестко кодировать в URL. Собственно, значение `templateUrl` должно содержать только имя файла без сегментов пути, как показано в листинге.

Обновление ссылок на модули

В новый модуль были перемещены два класса, команды `import` которых необходимо привести в соответствие с путями к другим модулям приложения. В листинге 21.25 показаны необходимые изменения в файле discountDisplay.component.ts.

Листинг 21.25. Обновление путей в файле discountDisplay.component.ts

```
import { Component, Input } from "@angular/core";
import { DiscountService } from "../common/discount.service";

@Component({
    selector: "paDiscountDisplay",
    template: `<div class="bg-info p-a-1">
                The discount is {{discounter.discount}}
               </div>`
})
export class PaDiscountDisplayComponent {

    constructor(private discounter: DiscountService) { }
}
```

В листинге 21.26 показаны те же изменения, примененные в файле discountEditor.component.ts.

Листинг 21.26. Обновление путей в файле discountEditor.component.ts

```
import { Component, Input } from "@angular/core";
import { DiscountService } from "../common/discount.service";

@Component({
    selector: "paDiscountEditor",
    template: `<div class="form-group">
                    <label>Discount</label>
                    <input [(ngModel)]="discounter.discount"
                         class="form-control" type="number" />
               </div>`
})
export class PaDiscountEditorComponent {

    constructor(private discounter: DiscountService) { }
}
```

Создание определения модуля

Чтобы создать модуль, создайте файл components.module.ts в папке app/components и добавьте в него команды из листинга 21.27.

Листинг 21.27. Содержимое файла components.module.ts в папке app/components

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { CommonModule } from "../common/common.module";
import { FormsModule, ReactiveFormsModule } from "@angular/forms"
import { PaDiscountDisplayComponent } from "./discountDisplay.component";
import { PaDiscountEditorComponent } from "./discountEditor.component";
import { ProductFormComponent } from "./productForm.component";
import { ProductTableComponent } from "./productTable.component";

@NgModule({
    imports: [BrowserModule, FormsModule, ReactiveFormsModule, CommonModule],
    declarations: [PaDiscountDisplayComponent, PaDiscountEditorComponent,
        ProductFormComponent, ProductTableComponent],
    exports: [ProductFormComponent, ProductTableComponent]
})
export class ComponentsModule { }
```

Модуль импортирует `BrowserModule` и `CommonModule`, чтобы директивы имели доступ к необходимым им службам и объявляемым классам. Он экспортирует `ProductFormComponent` и `ProductTableComponent` — два компонента, используемые в свойстве `bootstrap` корневого компонента. Остальные компоненты остаются приватными для модуля.

Обновление корневого модуля

Остается обновить корневой модуль, удалить устаревшие ссылки на отдельные файлы и импортировать новый модуль (листинг 21.28).

Листинг 21.28. Импортирование функционального модуля в файле app.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { ProductComponent } from "./component";
import { FormsModule, ReactiveFormsModule } from "@angular/forms";
import { ProductTableComponent } from "./components/productTable.component";
import { ProductFormComponent } from "./components/productForm.component";
import { ModelModule } from "./model/model.module";
import { CommonModule } from "./common/common.module";
import { ComponentsModule } from "./components/components.module";

@NgModule({
    imports: [BrowserModule, FormsModule, ReactiveFormsModule,
        ModelModule, CommonModule, ComponentsModule],
    bootstrap: [ProductFormComponent, ProductTableComponent]
})
export class AppModule { }
```

Добавление модулей в приложение радикально упрощает корневой модуль и позволяет определять взаимосвязанную функциональность в автономных блоках, которые можно расширять или модифицировать в относительной изоляции от остальных частей приложения.

Итоги

В этой главе была описана последняя разновидность структурных блоков Angular: модули. Я объяснил роль корневого модуля и показал, как создавать функциональные модули для структуризации приложения. В следующей главе будут описаны средства Angular для построения из этих структурных блоков сложных приложений, быстро реагирующих на действия пользователя.

22 Создание проекта

В предыдущих главах я добавлял в проект нашего примера классы и контент для демонстрации различных возможностей Angular. В главе 21 были добавлены функциональные модули, формирующие простейшую структуру проекта. В результате проект содержит множество избыточной и неиспользуемой функциональности, поэтому для этой главы будет создан новый проект, который заимствует ряд базовых возможностей из предыдущих глав. Он станет основой для построения следующих глав.

ПРИМЕЧАНИЕ

Полный проект включен в состав прилагаемого к книге архива с исходным кодом, который можно загрузить на сайте издательства apress.com.

Начало работы над проектом

Чтобы проект для этой главы отличался от предыдущих примеров, создайте папку с именем exampleApp, которая станет корневой папкой проекта. Я добавил серию вложенных папок, которые будут использоваться для хранения кода приложения и некоторых функциональных модулей (табл. 22.1).

ПРИМЕЧАНИЕ

Я не описываю структуру или назначение файлов, которые будут добавляться в проект в этой главе. За подробной информацией о том, как создаются проекты Angular, обращайтесь к главе 11.

Таблица 22.1. Папки, созданные для приложения

Имя	Описание
exampleApp	Корневая папка для всех файлов проекта
exampleApp/app	Папка, содержащая все файлы с программным кодом, модули, компоненты и шаблоны Angular
exampleApp/app/model	Папка для функционального модуля, содержащего модель данных

Имя	Описание
exampleApp/app/core	Папка для функционального модуля, содержащего компоненты с ключевой функциональностью приложения
exampleApp/app/messages	Папка для функционального модуля, используемого для вывода сообщений и информации об ошибках

Добавление и настройка пакетов

Добавьте файл с именем package.json в папку exampleApp и включите в него список пакетов, необходимых проекту, и сценарии NPM, которые будут использоваться для запуска инструментов разработки (листинг 22.1).

Листинг 22.1. Содержимое файла package.json в папке exampleApp

```
{
  "dependencies": {
    "@angular/common": "2.2.0",
    "@angular/compiler": "2.2.0",
    "@angular/core": "2.2.0",
    "@angular/platform-browser": "2.2.0",
    "@angular/platform-browser-dynamic": "2.2.0",
    "@angular/upgrade": "2.2.0",
    "@angular/forms": "2.2.0",
    "reflect-metadata": "0.1.8",
    "rxjs": "5.0.0-beta.12",
    "zone.js": "0.6.26",
    "core-js": "2.4.1",
    "classlist.js": "1.1.20150312",
    "systemjs": "0.19.40",
    "bootstrap": "4.0.0-alpha.4",
    "intl": "1.2.4"
  },

  "devDependencies": {
    "lite-server": "2.2.2",
    "typescript": "2.0.2",
    "typings": "1.3.2",
    "concurrently": "2.2.0",
    "systemjs-builder": "0.15.32"
  },

  "scripts": {
    "start": "concurrently \"npm run tscwatch\" \"npm run lite\" ",
    "tsc": "tsc",
    "tscwatch": "tsc -w",
    "lite": "lite-server",
    "typings": "typings",
    "postinstall": "typings install"
  }
}
```

Это те же пакеты, которые я использовал ранее, включая полизаполнение для API интернационализации, добавленное в главе 18. В разделе `devDependencies` появилась новая запись, которая будет использоваться в разделе «Создание модуля Reactive Extensions».

Чтобы загрузить и установить пакеты, выполните следующую команду из папки exampleApp:

```
npm install
```

Настройка TypeScript

После того как NPM загрузит и установит пакеты, выполните следующие команды из папки exampleApp, чтобы добавить информацию о типах, которая будет использоваться компилятором TypeScript:

```
npm run typings -- install dt~core-js --save --global
npm run typings -- install dt~node --save --global
```

Пакет `typings` получает последнюю информацию о типах для пакетов Core-JS и Node и сохраняет ее в папке с именем typings. Также будет сгенерирован файл typings.json, который содержит ссылки на пакеты (листинг 22.2).

Листинг 22.2. Содержимое файла typings.json в папке exampleApp

```
{
  "globalDependencies": {
    "core-js": "registry:dt/core-js#0.0.0+20160914114559",
    "node": "registry:dt/node#6.0.0+20161110151007"
  }
}
```

Также необходимо задать конфигурацию компилятора TypeScript, который будет генерировать код JavaScript, работающий в приложении Angular. Создайте файл tsconfig.json и добавьте свойства конфигурации из листинга 22.3. Это те же свойства, которые использовались в предыдущих главах книги.

Листинг 22.3. Содержимое файла tsconfig.json в папке exampleApp

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "moduleResolution": "node",
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true
  },
  "exclude": [ "node_modules" ]
}
```

Настройка сервера HTTP для разработки

Чтобы подготовить пакет BrowserSync, используемый сервером HTTP lite-server, создайте файл с именем bs-config.js в папке exampleApp и используйте его для определения конфигурации в листинге 22.4.

Листинг 22.4. Содержимое файла bs-config.js в папке exampleApp

```
module.exports = {
    ghostMode: false,
    reloadDelay: 1000,
    reloadDebounce: 1000,
    injectChanges: false,
    minify: false
}
```

Настройка загрузчика модулей JavaScript

Чтобы подготовить загрузчик модулей SystemJS, создайте файл systemjs.config.js в папке exampleApp и используйте его для определения конфигурации из листинга 22.5. Это та же конфигурация, которая использовалась в предыдущих главах.

Листинг 22.5. Содержимое файла systemjs.config.js в папке exampleApp

```
(function (global) {
    var paths = {
        "@angular/*": "node_modules/@angular/*"
    }

    var packages = { "app": {} };

    var angularModules = ["common", "compiler",
     "core", "platform-browser", "platform-browser-dynamic", "forms"];

    angularModules.forEach(function (pkg) {
        packages["@angular/" + pkg] = {
            main: "/bundles/" + pkg + ".umd.min.js"
        };
    });

    System.config({ paths: paths, packages: packages });

})(this);
```

Создание модуля модели

Первый функциональный модуль будет содержать модель данных проекта. Она похожа на ту, которая использовалась ранее, хотя и не содержат логики проверки данных формы (эта задача будет решаться в другом месте).

Создание типа данных Product

Чтобы определить основной тип данных для приложения, создайте файл с именем product.model.ts в папке exampleApp/app/model и включите в него определение класса из листинга 22.6.

Листинг 22.6. Файл product.model.ts в папке exampleApp/app/model

```
export class Product {

    constructor(public id?: number,
                public name?: string,
                public category?: string,
                public price?: number) {}
}
```

Создание источника данных и репозитория

Чтобы у приложения были исходные данные, создайте файл static.datasource.ts в папке exampleApp/app/model и включите определение службы из листинга 22.7. Этот класс будет использоваться как источник данных до главы 24, где я объясню, как использовать асинхронные запросы HTTP для получения данных от веб-служб.

ПРИМЕЧАНИЕ

При создании файлов функционального модуля я не так жестко следую правилам назначения имен файлов Angular, особенно если назначение модуля очевидно следует из его имени.

Листинг 22.7. Файл static.datasource.ts в папке exampleApp/app/model

```
import { Injectable } from "@angular/core";
import { Product } from "./product.model";

@Injectable()
export class StaticDataSource {
    private data: Product[];

    constructor() {
        this.data = new Array<Product>(
            new Product(1, "Kayak", "Watersports", 275),
            new Product(2, "Lifejacket", "Watersports", 48.95),
            new Product(3, "Soccer Ball", "Soccer", 19.50),
            new Product(4, "Corner Flags", "Soccer", 34.95),
            new Product(5, "Thinking Cap", "Chess", 16));
    }

    getData(): Product[] {
        return this.data;
    }
}
```

Следующим шагом должно стать определение репозитория, через который остальные части приложения будут обращаться к модели данных. Создайте файл repository.model.ts в папке exampleApp/app/model и используйте его для определения класса из листинга 22.8.

Листинг 22.8. Файл repository.model.ts в папке exampleApp/app/model

```
import { Injectable } from "@angular/core";
import { Product } from "./product.model";
import { StaticDataSource } from "./static.datasource";

@Injectable()
export class Model {
    private products: Product[];
    private locator = (p: Product, id: number) => p.id == id;

    constructor(private dataSource: StaticDataSource) {
        this.products = new Array<Product>();
        this.dataSource.getData().forEach(p => this.products.push(p));
    }

    getProducts(): Product[] {
        return this.products;
    }

    getProduct(id: number): Product {
        return this.products.find(p => this.locator(p, id));
    }

    saveProduct(product: Product) {
        if (product.id == 0 || product.id == null) {
            product.id = this.generateID();
            this.products.push(product);
        } else {
            let index = this.products
                .findIndex(p => this.locator(p, product.id));
            this.products.splice(index, 1, product);
        }
    }

    deleteProduct(id: number) {
        let index = this.products.findIndex(p => this.locator(p, id));
        if (index > -1) {
            this.products.splice(index, 1);
        }
    }

    private generateID(): number {
        let candidate = 100;
        while (this.getProduct(candidate) != null) {
            candidate++;
        }
        return candidate;
    }
}
```

Завершение модуля модели

Чтобы завершить модель данных, необходимо определить модуль. Создайте файл model.module.ts в папке exampleApp/app/model и включите в него определение модуля Angular из листинга 22.9.

Листинг 22.9. Содержимое файла model.module.ts в папке exampleApp/app/model

```
import { NgModule } from "@angular/core";
import { StaticDataSource } from "./static.datasource";
import { Model } from "./repository.model";

@NgModule({
    providers: [Model, StaticDataSource]
})
export class ModelModule { }
```

Создание базового модуля

Базовый модуль будет содержать центральную функциональность приложения, построенную на основе возможностей, описанных в главе 2: вывод списка товаров в модели для пользователя с возможностями их создания и редактирования.

Создание службы общего состояния

Чтобы обеспечить взаимодействие компонентов этого модуля, мы добавим службу, которая сохраняет текущий режим (редактирование или создание товара). Добавьте файл sharedState.model.ts в папку exampleApp/app/core и включите в него определения перечисления и класса из листинга 22.10.

ПРИМЕЧАНИЕ

Я использовал имя файла model.ts вместо service.ts, потому что в дальнейших главах роль этого класса изменится. А пока будьте ко мне снисходительны, хотя я и нарушаю правила назначения имен.

Листинг 22.10. Содержимое файла sharedState.model.ts в папке exampleApp/app/core

```
export enum MODES {
    CREATE, EDIT
}

export class SharedState {
    mode: MODES = MODES.EDIT;
    id: number;
}
```

Класс `SharedState` содержит два свойства: для текущего режима и идентификатора объекта модели данных, с которым работает приложение.

Создание компонента таблицы

Этот компонент выводит таблицу со списком всех товаров. Таблица занимает центральное место в приложении: она предоставляет доступ к другим функциональным областям при помощи кнопок создания, редактирования и удаления объектов. В листинге 22.11 показано содержимое файла table.component.ts в папке exampleApp/app/core.

Листинг 22.11. Содержимое файла table.component.ts в папке exampleApp/app/core

```
import { Component } from "@angular/core";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
import { MODES, SharedState } from "./sharedState.model";

@Component({
    selector: "paTable",
    moduleId: module.id,
    templateUrl: "table.component.html"
})
export class TableComponent {

    constructor(private model: Model, private state: SharedState) { }

    getProduct(key: number): Product {
        return this.model.getProduct(key);
    }

    getProducts(): Product[] {
        return this.model.getProducts();
    }

    deleteProduct(key: number) {
        this.model.deleteProduct(key);
    }

    editProduct(key: number) {
        this.state.id = key;
        this.state.mode = MODES.EDIT;
    }

    createProduct() {
        this.state.id = undefined;
        this.state.mode = MODES.CREATE;
    }
}
```

Компонент предоставляет базовую функциональность, использованную в предыдущих главах, с добавлением методов `editProduct` и `createProduct`. Эти методы обновляют службу общего состояния, когда пользователь захочет отредактировать или создать товар.

Создание шаблона компонента таблицы

Чтобы предоставить шаблон для компонента таблицы, добавьте файл HTML с именем table.component.html в папку exampleApp/app/core и включите в него разметку из листинга 22.12.

Листинг 22.12. Файл table.component.html в папке exampleApp/app/core

```
<table class="table table-sm table-bordered table-striped">
    <tr>
        <th>ID</th><th>Name</th><th>Category</th><th>Price</th><th></th>
    </tr>
    <tr *ngFor="let item of getProducts()">
        <td style="vertical-align:middle">{{item.id}}</td>
        <td style="vertical-align:middle">{{item.name}}</td>
        <td style="vertical-align:middle">{{item.category}}</td>
        <td style="vertical-align:middle">
            {{item.price | currency:"USD":true }}
        </td>
        <td class="text-xs-center">
            <button class="btn btn-danger btn-sm" (click)="deleteProduct(item.id)">
                Delete
            </button>
            <button class="btn btn-warning btn-sm" (click)="editProduct(item.id)">
                Edit
            </button>
        </td>
    </tr>
</table>
<button class="btn btn-primary" (click)="createProduct()">
    Create New Product
</button>
```

Шаблон использует директиву `ngFor` для создания в таблице строки для каждого товара в модели данных вместе с кнопками для вызова методов `deleteProduct` и `editProduct`. Также за пределами таблицы находится элемент `button`, при щелчке на котором вызывается метод `createProduct` компонента.

Создание компонента формы

В этом проекте мы создадим компонент формы, который будет управлять формой HTML для создания новых и редактирования существующих продуктов. Чтобы создать определение компонента, добавьте файл form.component.ts в папку exampleApp/app/core и включите в него код из листинга 22.13.

Листинг 22.13. Содержимое файла form.component.ts в папке exampleApp/app/core

```
import { Component } from "@angular/core";
import { NgForm } from "@angular/forms";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model"
import { MODES, SharedState } from "./sharedState.model";
```

```
@Component({
    selector: "paForm",
    moduleId: module.id,
    templateUrl: "form.component.html",
    styleUrls: ["form.component.css"]
})
export class FormComponent {
    product: Product = new Product();
    constructor(private model: Model,
            private state: SharedState) { }

    get editing(): boolean {
        return this.state.mode == MODES.EDIT;
    }

    submitForm(form: NgForm) {
        if (form.valid) {
            this.model.saveProduct(this.product);
            this.product = new Product();
            form.reset();
        }
    }
    resetForm() {
        this.product = new Product();
    }
}
```

Один компонент и форма будут использоваться как для создания новых, так и для редактирования существующих товаров, поэтому добавляется новая функциональность. Существующее свойство будет использоваться в представлении для обозначения текущего режима службы общего состояния. Метод `resetForm` (еще одно новшество) сбрасывает состояние объекта, предоставляющего данные для формы. Метод `submitForm` не изменился; он использует модель данных для определения того, является объект, переданный методу `saveProduct`, новым или же заменяет существующий объект.

Создание шаблона для компонента формы

Чтобы создать шаблон для компонента, создайте файл HTML с именем form.component.html в папке exampleApp/app/core и добавьте в него разметку из листинга 22.14.

Листинг 22.14. Файл form.component.html в папке exampleApp/app/core

```
<div class="bg-primary p-a-1" [class.bg-warning]="editing">
    <h5>{{editing  ? "Edit" : "Create"}} Product</h5>
</div>

<form novalidate #form="ngForm" (ngSubmit)="submitForm(form)"
                                (reset)="resetForm()" >
```

```
    <div class="form-group">
        <label>Name</label>
        <input class="form-control" name="name"
               [(ngModel)]="product.name" required />
    </div>

    <div class="form-group">
        <label>Category</label>
        <input class="form-control" name="category"
               [(ngModel)]="product.category" required />
    </div>

    <div class="form-group">
        <label>Price</label>
        <input class="form-control" name="price"
               [(ngModel)]="product.price"
               required pattern="^[0-9\.]+$" />
    </div>

    <button type="submit" class="btn btn-primary"
            [class.btn-warning]="editing" [disabled]="form.invalid">
        {{editing ? "Save" : "Create"}}
    </button>
    <button type="reset" class="btn btn-secondary">Cancel</button>
</form>
```

Самая важная часть этого шаблона — форма — содержит элементы `input` для свойств имени, категории и цены, необходимых для создания или редактирования товара. Заголовок в верхней части шаблона и кнопка отправки данных на форме изменяют свой контент и внешний вид в зависимости от режима редактирования, чтобы пользователь мог различать выполняемые операции.

Создание стилей для компонента формы

Чтобы не усложнять пример, я использовал базовую проверку формы без сообщений об ошибках. Вместо них я использую стили CSS, применяемые к классам проверки данных Angular. Создайте файл form.component.css в папке exampleApp/app/core и включите в него определения стилей из листинга 22.15.

Листинг 22.15. Файл form.component.css в папке exampleApp/app/core

```
input.ng-dirty.ng-invalid { border: 2px solid #ff0000 }
input.ng-dirty.ng-valid { border: 2px solid #6bc502 }
```

Создание базового модуля

Чтобы определить модуль, содержащий компоненты, создайте файл core.module.ts в папке exampleApp/app/core и включите в него определение модуля Angular из листинга 22.16.

Листинг 22.16. Файл core.module.ts в папке exampleApp/app/core

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule } from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { TableComponent } from "./table.component";
import { FormComponent } from "./form.component";
import { SharedState } from "./sharedState.model";

@NgModule({
    imports: [BrowserModule, FormsModule, ModelModule],
    declarations: [TableComponent, FormComponent],
    exports: [ModelModule, TableComponent, FormComponent],
    providers: [SharedState]
})
export class CoreModule { }
```

Модуль импортирует три модуля для использования базовой функциональности Angular, поддержки форм Angular и модели данных приложения (эти модули были созданы ранее в этой главе). Также он определяет провайдера для службы `SharedState`.

Создание модуля сообщений

Модуль сообщений будет содержать службу, которая используется для выдачи информационных сообщений или ошибок, а также компонент, в котором эти сообщения будут отображаться. Эта функциональность будет использоваться во всем приложении и не принадлежит ни одному из двух других модулей.

Создание модели сообщения и службы

Для представления выводимых сообщений создайте файл message.model.ts в папке exampleApp/app/messages и добавьте в него код из листинга 22.17.

Листинг 22.17. Содержимое файла message.model.ts в папке exampleApp/app/messages

```
export class Message {

    constructor(private text: string,
        private error: boolean = false) { }
}
```

Свойства класса `Message` представляют текст, выводимый для пользователя, а также признак ошибки. Затем создайте файл message.service.ts в папке exampleApp/app/messages и включите в него определение службы из листинга 22.18. Служба будет использоваться для регистрации сообщений, выводимых для пользователя.

Листинг 22.18. Содержимое файла message.service.ts в папке exampleApp/app/messages

```
import { Injectable } from "@angular/core";
import { Message } from "./message.model";

@Injectable()
export class MessageService {
    private handler: (m: Message) => void;

    reportMessage(msg: Message) {
        if (this.handler != null) {
            this.handler(msg);
        }
    }

    registerMessageHandler(handler: (m: Message) => void) {
        this.handler = handler;
    }
}
```

Служба играет роль посредника между частями приложения, порождающими сообщения об ошибках, и теми, которые эти сообщения должны получать. Служба будет усовершенствована в главе 23, когда я представлю возможности пакета Reactive Extensions.

Создание компонента и шаблона

Теперь, когда у нас имеется источник сообщений, мы можем создать компонент для их вывода. Создайте файл message.component.ts в папке exampleApp/app/messages и включите определение компонента из листинга 22.19.

Листинг 22.19. Содержимое файла message.component.ts в папке exampleApp/app/messages

```
import { Component } from "@angular/core";
import { MessageService } from "./message.service";
import { Message } from "./message.model";

@Component({
    selector: "paMessages",
    moduleId: module.id,
    templateUrl: "message.component.html",
})
export class MessageComponent {
    lastMessage: Message;

    constructor(messageService: MessageService) {
        messageService.registerMessageHandler(m => this.lastMessage = m);
    }
}
```

Компонент получает объект `MessageService` в аргументе конструктора и использует его для регистрации функции-обработчика, которая будет вызываться при получении сообщения службой. Самое последнее сообщение будет сохранено в свойстве `lastMessage`. Чтобы предоставить шаблон для компонента, создайте файл message.component.html в папке exampleApp/app/messages и добавьте в него разметку для вывода сообщения из листинга 22.20.

Листинг 22.20. Файл message.component.html в папке exampleApp/app/messages

```
<div *ngIf="lastMessage"
     class="bg-info p-a-1 text-xs-center"
     [class.bg-danger]="lastMessage.error">
        <h4>{{lastMessage.text}}</h4>
</div>
```

Завершение модуля сообщений

Создайте файл message.module.ts в папке exampleApp/app/messages и включите в него определение модуля из листинга 22.21.

Листинг 22.21. Файл message.module.ts в папке exampleApp/app/messages

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { MessageComponent } from "./message.component";
import { MessageService } from "./message.service";

@NgModule({
    imports: [BrowserModule],
    declarations: [MessageComponent],
    exports: [MessageComponent],
    providers: [MessageService]
})
export class MessageModule { }
```

Завершение проекта

Чтобы создать в проекте корневой модуль, создайте файл app.module.ts в папке exampleApp/app и включите в него определение модуля Angular из листинга 22.22.

Листинг 22.22. Содержимое файла app.module.ts в папке exampleApp/app

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { ModelModule } from "./model/model.module";
import { CoreModule } from "./core/core.module";
import { TableComponent } from "./core/table.component";
import { FormComponent } from "./core/form.component";
import { MessageModule } from "./messages/message.module";
import { MessageComponent } from "./messages/message.component";

@NgModule({
```

```
    imports: [BrowserModule, ModelModule, CoreModule, MessageModule],
    bootstrap: [TableComponent, FormComponent, MessageComponent]
})
export class AppModule { }
```

Этот модуль импортирует функциональные модули, созданные в этой главе, а также модуль `BrowserModule`, предоставляющий доступ к базовой функциональности Angular. Модуль задает три корневых компонента; два из них были определены в `CoreModule`, а один в `MessageModule`. Эти компоненты выводят таблицу товаров, форму и информационные сообщения/ошибки.

Создание файла начальной загрузки Angular

Создайте файл main.ts в папке exampleApp/app. Добавьте в него команды из листинга 22.23.

Листинг 22.23. Содержимое файла main.ts в папке exampleApp/app

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

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

Вызов метода `bootstrapModule` приказывает Angular использовать класс, определенный в листинге 22.22, в качестве корневого модуля приложения.

Создание модуля Reactive Extensions

В предыдущей части книги я настроил загрузчик модулей JavaScript SystemJS так, чтобы он разрешал все зависимости пакета `rxjs`, загружая файл node_modules/rxjs/bundles/Rx.min.js. Такое решение хорошо работает — пока вы не начнете пользоваться функциональностью, предоставляемой Reactive Extensions, прямо в своем коде (как в главе 23) или же использовать модуль маршрутизации Angular (см. главу 25).

Проблема заключается в том, что пакет Reactive Extensions не содержит информации, необходимой загрузчику SystemJS для загрузки и обработки модуля, — для ее решения придется сгенерировать специальный файл модуля. Пакет `systemjs-builder`, который я добавил в проект в файле package.json, используется для построения и настройки конфигурации модулей. Создайте файл rxModuleBuilder.js в папке exampleApp и добавьте в него код из листинга 22.24.

Листинг 22.24. Содержимое файла rxModuleBuilder.js в папке exampleApp

```
var Builder = require("systemjs-builder");

var builder = new Builder("./");
builder.config({
    paths: {
        "rxjs/*": "node_modules/rxjs/*.js"
    },
```

```
    map: {
        "rxjs": "node_modules/rxjs"
    },
    packages: {
        "rxjs": { main: "Rx.js", defaultExtension: "js" }
    }
});
builder.bundle("rxjs", "rxjs.module.min.js", {
    normalize: true,
    runtime: false,
    minify: true,
    mangle: false
});
```

Эта конфигурация создает модуль, содержащий всю функциональность Reactive Extensions, вместе с дополнительной информацией, необходимой SystemJS для разрешения зависимостей. Выполните следующую команду из папки exampleApp:

```
node ./rxModuleBuilder.js
```

Обработка пакета может занять несколько минут, но после ее завершения в папке exampleApp появляется файл с именем rxjs.module.min.js. Этот файл будет использоваться в проекте для работы с пакетом Reactive Extensions.

Создание документа HTML

Чтобы завершить приложение, создайте файл index.html в папке exampleApp и добавьте в него разметку HTML из листинга 22.25.

Листинг 22.25. Содержимое файла index.html в папке exampleApp

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
    <meta charset="utf-8" />
    <script src="node_modules/classlist.js/classList.min.js"></script>
    <script src="node_modules/core-js/client/shim.min.js"></script>
    <script src="node_modules/intl/dist/Intl.complete.js"></script>
    <script src="node_modules/zone.js/dist/zone.min.js"></script>
    <script src="node_modules/reflect-metadata/Reflect.js"></script>
    <script src="node_modules/systemjs/dist/system.src.js"></script>
    <script src="rxjs.module.min.js"></script>
    <script src="systemjs.config.js"></script>
    <script>
      System.import("app/main").catch(function(err){ console.error(err); });
    </script>
    <link href="node_modules/bootstrap/dist/css/bootstrap.min.css"
        rel="stylesheet" />
</head>
<body class="m-a-1">
    <paMessages></paMessages>
```

```
    <div class="col-xs-8 p-a-1">
        <paTable></paTable>
    </div>
    <div class="col-xs-4 p-a-1">
        <paForm></paForm>
    </div>
</body>
</html>
```

Элементы `paTable` и `paForm` соответствуют селекторам компонентов, определенным в базовом функциональном модуле, и размещаются рядом друг с другом с использованием классов Bootstrap. Элемент `paMessages` применяет компонент из модуля `messages`.

ПРИМЕЧАНИЕ

Обратите внимание на элемент script для файла rxjs.module.min.js, который был создан в предыдущем разделе. При загрузке файла с использованием элемента script используется информация, сгенерированная пакетом systemjs-builder.

Запуск приложения

Чтобы запустить компилятор TypeScript и сервер HTTP для разработки, выполните следующую команду из папки exampleApp:

```
npm start
```

Код TypeScript компилируется в JavaScript, а сервер для разработки открывает новую вкладку или окно браузера с контентом, показанным на рис. 22.1.

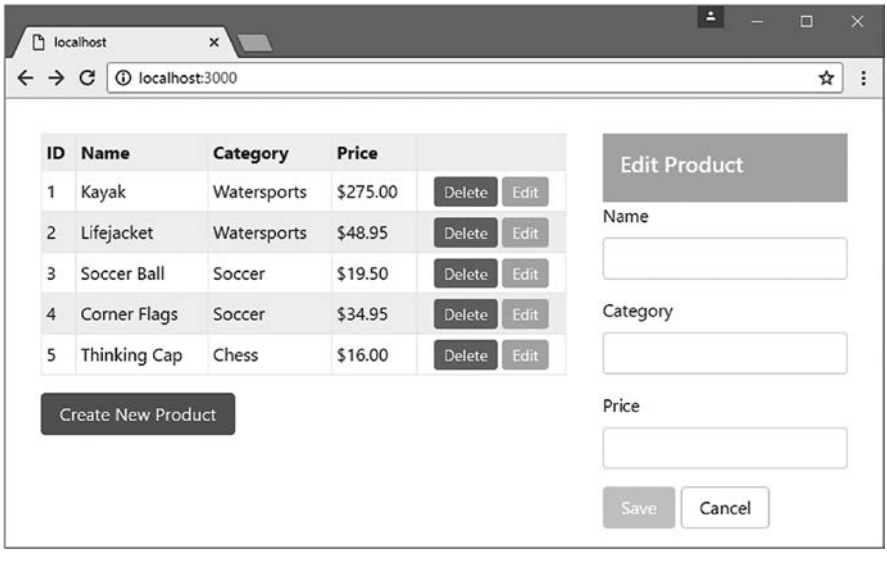


Рис. 22.1. Запуск приложения

Пока работают не все части приложения. Вы можете переключаться между двумя режимами работы кнопками Create New Product и Edit, но функция редактирования не работает. Базовая функциональность и добавление новых возможностей будут реализованы в следующих главах.

Итоги

В этой главе мы создали проект, который будет использоваться далее. Базовая структура напоминает ту, что использовалась в предыдущих главах, но в проекте нет лишнего кода и разметки, которые использовались ранее для демонстрации. В следующей главе рассматривается пакет Reactive Extensions, обеспечивающий обработку обновлений в приложениях Angular.

23 Reactive Extensions

У Angular много интересных возможностей, но наибольшего внимания заслуживает механизм распространения изменений в приложении, благодаря которому заполнение поля формы или нажатие кнопки немедленно приводит к обновлению состояния приложения. Однако изменения, отслеживаемые Angular, ограничены, и функции требуют прямой работы с библиотекой, которую Angular использует для распространения обновлений в приложении. Эта библиотека называется *Reactive Extensions*, или сокращенно *RxJS*.

В этой главе я объясню, почему работа с Reactive Extensions практически обязательна для нетривиальных проектов, представлю ключевые возможности Reactive Extensions (`Observer` и `Observable`) и воспользуюсь ими для расширения приложения, чтобы пользователи могли изменять существующие объекты в модели, а также создавать новые объекты.

В табл. 23.1 библиотека Reactive Extensions представлена в контексте.

ПРИМЕЧАНИЕ

Эта глава посвящена возможностям RxJS, приносящим наибольшую пользу в проектах Angular. Пакет RxJS обладает широкой функциональностью, и если вам понадобится дополнительная информация, обратитесь на домашнюю страницу проекта по адресу https://github.com/Reactive-Extensions/RxJS.

Таблица 23.1. Reactive Extensions в контексте

Вопрос	Ответ
Что это такое?	Библиотека Reactive Extensions реализует механизм асинхронного распространения событий, широко применяемый в Angular для обнаружения изменений и распространения событий
Для чего они нужны?	RxJS позволяет частям приложения, не входящим в стандартный процесс обнаружения изменений Angular, получать уведомления о важных событиях и реагировать на них. Так как библиотека RxJS необходима для использования Angular, ее функциональность уже доступна для разработчика

Вопрос	Ответ
Как они используются?	Объект Observer собирает события и распространяет их среди подписчиков через Observable. В простейшем решении создается объект Subject, который предоставляет функциональность как Observer, так и Observable. Для управления потоком событий к подписчику используются специальные операторы
Есть ли у них недостатки или скрытые проблемы?	После того как вы освоите базовые возможности, работать с пакетом RxJS несложно. Впрочем, из-за широты возможностей иногда приходится экспериментировать, чтобы обнаружить комбинацию для эффективного достижения желаемого результата
Есть ли альтернативы?	Пакет RxJS необходим для использования некоторых функций Angular, таких как обновление потомков, получение информации о потомках и выдача асинхронных запросов HTTP

В табл. 23.2 приведена краткая сводка материала главы.

Таблица 23.2. Сводка материала главы

Проблема	Решение	Листинг
Распространение событий в приложении	Используйте Reactive Extensions	1–5
Ожидание асинхронных результатов в шаблоне	Используйте канал async	6–9
Использование событий для взаимодействия между компонентами	Используйте Observable	10–12
Управление потоком событий	Используйте такие операторы, как filter и map	13–18

Подготовка проекта

В этой главе используется проект exampleApp, созданный в главе 22. Никакая дополнительная подготовка для этой главы не потребуется.

Выполните следующую команду из папки exampleApp, чтобы запустить компилятор TypeScript и сервер HTTP для разработки:

```
npm start
```

Открывается новая вкладка или окно браузера с контентом, показанным на рис. 23.1.

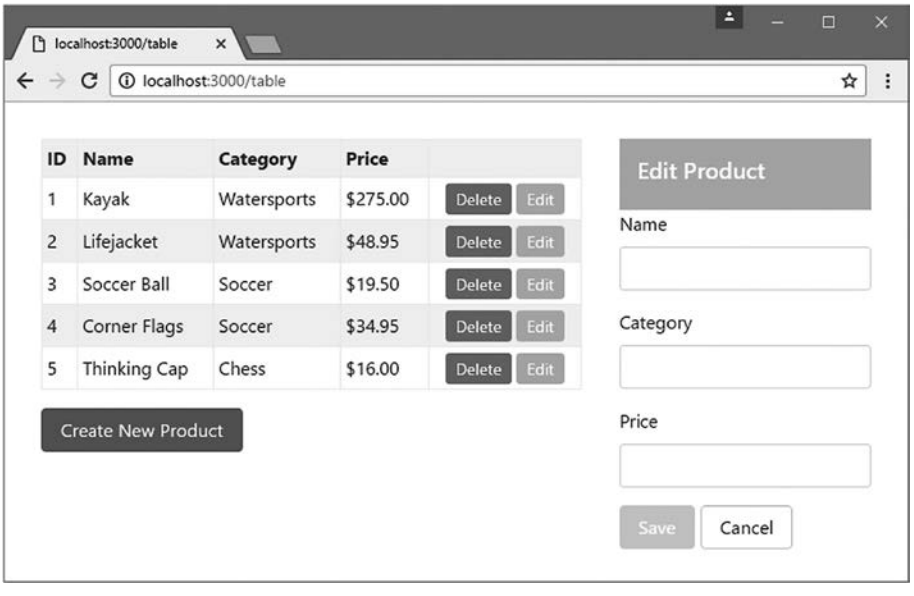


Рис. 23.1. Запуск приложения

ПРИМЕЧАНИЕ

Если вы не хотите создавать этот проект шаг за шагом, его (а также проекты всех остальных глав книги) можно взять из бесплатного архива кода, прилагаемого к книге, на сайте издательства apress.com.

Суть проблемы

Angular отлично справляется с обнаружением изменений в выражениях привязки данных. Эта задача решается эффективно и естественно, а в результате формируется инфраструктура, упрощающая создание динамических приложений. Чтобы понаблюдать за механизмом обнаружения изменений в деле, щелкните на кнопке Create New Product. Служба, предоставляющая информацию общего состояния, обновляется компонентом таблицы; затем она отражается в привязках данных, управляемых компонентом формы (рис. 23.2). При щелчке на кнопке Create New Product цвета заголовков и кнопок на форме немедленно изменяются.

С ростом числа объектов в приложении механизм обнаружения изменений выходит из-под контроля и начинает катастрофически снижать быстродействие приложения, особенно на менее мощных устройствах (таких, как телефоны и планшеты). Вместо того чтобы отслеживать все объекты в приложении, Angular концентрируется на привязках данных, а конкретнее — на изменениях значений свойств.

И это создает новую проблему, потому что Angular управляет привязками элементов HTML автоматически, но не предоставляет поддержки для обработки изменений в самом компоненте. Чтобы увидеть прямое следствие отсутствия изменений

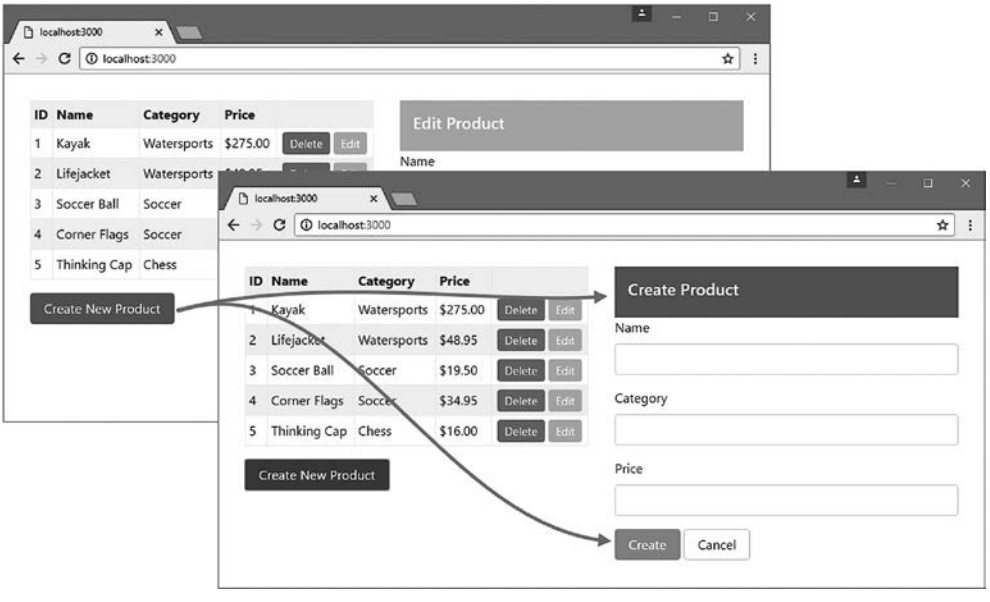


Рис. 23.2. Обновление выражений привязки данных

в компоненте, щелкните на одной из кнопок Edit в таблице. Хотя привязки данных немедленно обновляются, компонент не получает оповещения при щелчке на кнопке и не знает, что ему нужно обновить свойство, которое заполнит элементы формы для редактирования.

Отсутствие обновлений означает, что для выявления важных изменений компонент формы должен вернуться к методу ngDoCheck из главы 15, как показано в листинге 23.1.

Листинг 23.1. Отслеживание изменений службы в файле form.component.ts

```
import { Component } from "@angular/core";
import { NgForm } from "@angular/forms";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model"
import { MODES, SharedState } from "./sharedState.model";

@Component({
    selector: "paForm",
    moduleId: module.id,
    templateUrl: "form.component.html",
    styleUrls: ["form.component.css"]
})
export class FormComponent {
    product: Product = new Product();
    lastId: number;

    constructor(private model: Model,
```

```
        private state: SharedState) { }

    get editing(): boolean {
        return this.state.mode == MODES.EDIT;
    }

    submitForm(form: NgForm) {
        if (form.valid) {
            this.model.saveProduct(this.product);
            this.product = new Product();
            form.reset();
        }
    }

    resetForm() {
        this.product = new Product();
    }

    ngDoCheck() {
        if (this.lastId != this.state.id) {
            this.product = new Product();
            if (this.state.mode == MODES.EDIT) {
                Object.assign(this.product, this.model.getProduct(this.state.id));
            }
            this.lastId = this.state.id;
        }
    }
}
```

Чтобы понаблюдать за последствиями изменений, щелкните на одной из кнопок Edit в таблице; форма заполняется данными для редактирования. Когда вы завершите редактирование значений на форме, щелкните на кнопке Save; модель данных обновляется в соответствии с изменениями в таблице (рис. 23.3).

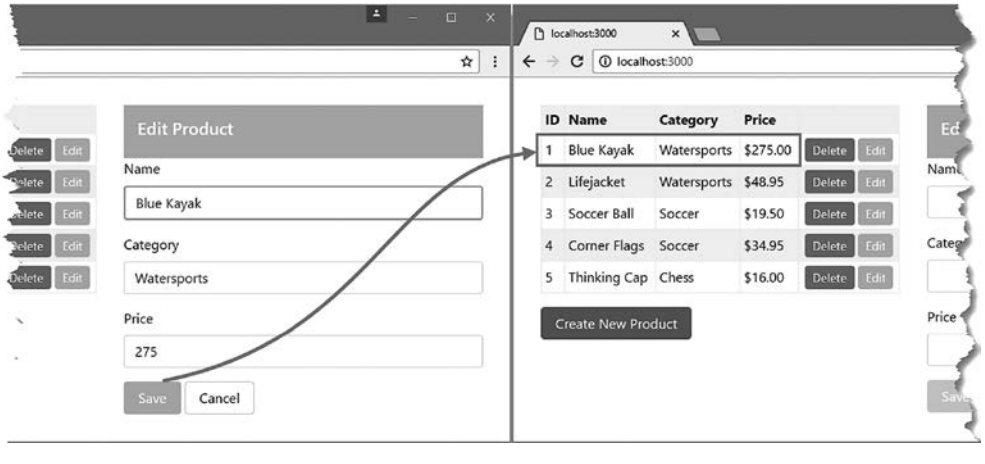


Рис. 23.3. Обновление продукта

Проблема этого кода заключается в том, что метод ngDoCheck будет вызываться каждый раз, когда Angular обнаружит какие-либо изменения в приложении. Неважно, что и где происходит: Angular все равно вызовет метод ngDoCheck, чтобы компонент имел возможность обновиться. Вы можете свести к минимуму объем работы, выполняемой в методе ngDoCheck, но с ростом количества директив и компонентов в приложении количество событий изменения и количество вызовов метода ngDoCheck растет, что может привести к снижению производительности приложения.

Кроме того, правильно обрабатывать изменения сложнее, чем можно подумать. Например, попробуйте отредактировать товар в приложении, щелкнуть на кнопке Save для сохранения изменений в модели, а затем снова щелкнуть на кнопке Edit для того же товара: ничего не произойдет. Это распространенная ошибка в реализации метода ngDoCheck, который вызывается даже в том случае, если изменение инициировано самим компонентом; они сбивают с толку проверки в методе ngDoCheck, которые пытаются предотвратить лишнюю работу. В целом такое решение ненадежно, затратно и плохо масштабируется.

Решение проблемы при помощи Reactive Extensions

Библиотека Reactive Extensions в приложениях Angular предоставляет простую, ясную систему отправки и получения уведомлений. На первый взгляд достижение не такое уж значительное, но оно лежит в основе большинства встроенных функций Angular и может применяться напрямую в приложениях для предотвращения проблем, возникающих при попытке реализовать обнаружение изменений с использованием ngDoCheck.

Чтобы подготовиться к прямой работе с Reactive Extensions, листинг 23.2 определяет маркер, который будет использоваться для предоставления службы, использующей Reactive Extensions для распространения обновлений, а также изменяет класс SharedState, чтобы он определял конструктор. Эти изменения временно нарушают работоспособность приложения, потому что Angular не сможет предоставлять значения конструктору SharedState при попытке создания экземпляра, который будет использоваться в качестве службы. Приложение снова заработает после внесения всех изменений, необходимых для Reactive Extensions.

Листинг 23.2. Определение маркера провайдера в файле sharedState.model.ts

```
import { OpaqueToken } from "@angular/core";

export enum MODES {
    CREATE, EDIT
}

export class SharedState {

    constructor(public mode: MODES, public id?: number) { }
}

export const SHARED_STATE = new OpaqueToken("shared_state");
```

Объекты Observable

Ключевой структурный блок Reactive Extensions — класс `Observable` — представляет последовательность наблюдаемых (отслеживаемых) в программе событий. Объект (например, компонент) может подписаться на `Observable` и получать уведомление при каждом наступлении события. Это позволяет ему реагировать только на наблюдаемые события (а не на изменения в любом месте приложения).

Основной метод `Observable` — `subscribe` — получает в аргументах три функции, описанные в табл. 23.3.

Таблица 23.3. Аргументы метода subscribe

Имя	Описание
onNext	Функция вызывается при инициировании нового события
onError	Функция вызывается при инициировании ошибки
onCompleted	Функция вызывается при завершении последовательности событий

Для подписки на `Observable` необходима только функция `onNext`, хотя обычно рекомендуется реализовать другие функции для обработки ошибок и реакции на возможное завершение последовательности событий. В этом примере конца серии событий не будет, но в других ситуациях с использованием `Observable` (например, при обработке ответов HTTP) информация о завершении последовательности событий может быть более полезной. В листинге 23.3 приведена переработанная версия компонента формы с объявлением зависимости от службы `Observable`.

Листинг 23.3. Использование Observable в файле form.component.ts

```
import { Component, Inject } from "@angular/core";
import { NgForm } from "@angular/forms";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
import { MODES, SharedState, SHARED_STATE } from "./sharedState.model";
import { Observable } from "rxjs/Observable";

@Component({
    selector: "paForm",
    moduleId: module.id,
    templateUrl: "form.component.html",
    styleUrls: ["form.component.css"]
})
export class FormComponent {
    product: Product = new Product();

    constructor(private model: Model,
        @Inject(SHARED_STATE) private stateEvents: Observable<SharedState>) {

        stateEvents.subscribe((update) => {
            this.product = new Product();
            if (update.id != undefined) {
                Object.assign(this.product, this.model.getProduct(update.id));
```

```
            }
            this.editing = update.mode == MODES.EDIT;
        });
    }

    editing: boolean = false;

    submitForm(form: NgForm) {
        if (form.valid) {
            this.model.saveProduct(this.product);
            this.product = new Product();
            form.reset();
        }
    }

    resetForm() {
        this.product = new Product();
    }
}
```

NPM-пакет Reactive Extensions включает отдельные модули JavaScript для всех предоставляемых типов, чтобы вы могли импортировать тип `Observable` из модуля `rxjs/Observable`.

Чтобы получать уведомления, компонент объявляет зависимость от службы `SHARED_STATE`, которая передается как объект `Observable<SharedState>`. Это объект `Observerable`, уведомлениями которого будут объекты `SharedState` и который будет представлять операцию редактирования или создания товара, инициированную пользователем. Компонент вызывает метод `Observable.subscribe`, передавая ему функцию, которая получает каждый объект `SharedState` и использует его для обновления состояния.

А КАК ЖЕ PROMISE?

Возможно, вы привыкли к представлению асинхронных операций с использованием объектов Promise. Объекты Observable делают приблизительно то же самое, но обладают большей гибкостью и более широкой функциональностью. Angular предоставляет поддержку работы с объектами Promise, которая может пригодиться при переходе на Angular и при работе с библиотеками, зависящими от Promise (например, jQuery).

Библиотека Reactive Extensions предоставляет метод `Observable.fromPromise`, который создает объект Observable, использующий Promise как источник событий. Также поддерживается метод Observable.toPromise, если у вас имеется объект Observable, а вам по каким-то причинам нужен Promise.

Кроме того, некоторые функциональные возможности Angular позволяют выбрать, какие объекты вы будете использовать. Например, механизм стражей (guards), описанный в главе 27, поддерживает объекты обоих видов.

Но библиотека Reactive Extensions является важной частью работы с Angular, и она будет часто встречаться вам в следующих главах. Я рекомендую работать с Observable везде, где это возможно, и свести к минимуму преобразования в Promise и обратно.

Объекты Observer

Объект Reactive Extensions `Observer` предоставляет механизм создания обновлений. Используемые при этом методы перечислены в табл. 23.4.

Таблица 23.4. Методы Observer

Имя	Описание
next(value)	Метод создает новое событие с использованием заданного значения value
error(errorObject)	Метод сообщает об ошибке, описываемой при помощи аргумента, в котором может передаваться произвольный объект
complete()	Метод завершает последовательность и сообщает о том, что события отправляться больше не будут

Листинг 23.4 обновляет компонент таблицы, чтобы он использовал `Observer` для отправки событий, когда пользователь щелкает на кнопке Create New Product или одной из кнопок Edit.

Листинг 23.4. Использование Observer в файле table.component.ts

```
import { Component, Inject } from "@angular/core";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
import { MODES, SharedState, SHARED_STATE } from "./sharedState.model";
import { Observer } from "rxjs/Observer";

@Component({
    selector: "paTable",
    moduleId: module.id,
    templateUrl: "table.component.html"
})
export class TableComponent {

    constructor(private model: Model,
        @Inject(SHARED_STATE) private observer: Observer<SharedState>) { }

    getProduct(key: number): Product {
        return this.model.getProduct(key);
    }

    getProducts(): Product[] {
        return this.model.getProducts();
    }

    deleteProduct(key: number) {
        this.model.deleteProduct(key);
    }
```

```
    editProduct(key: number) {
        this.observer.next(new SharedState(MODES.EDIT, key));
    }

    createProduct() {
        this.observer.next(new SharedState(MODES.CREATE));
    }
}
```

Компонент объявляет зависимость от службы `SHARED_STATE`, которая передается в виде объекта `Observer<SharedState>`; это означает, что `Observer` будет отправлять события, описываемые объектами `SharedState`. Методы `editProduct` и `createProduct` были обновлены, чтобы они вызывали методы `Observer` для передачи сигнала об изменении состояния.

Объекты Subject

Оба компонента объявляют зависимости от служб с использованием маркера `SHARED_STATE`, но они рассчитывают получать разные типы: компонент таблицы ожидает получать объект `Observer<SharedState>`, а компоненту формы нужен объект `Observable<SharedState>`.

Библиотека Reactive Extensions предоставляет класс `Subject`, реализующий функциональность как `Observer`, так и `Observable`. Благодаря этому легко создавать службы, позволяющие генерировать и потреблять события в одном объекте. В листинге 23.5 служба, объявленная в свойстве `providers` декоратора `@NgModule`, изменяется для использования объекта `Subject`.

Листинг 23.5. Изменение службы в файле core.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule } from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { TableComponent } from "./table.component";
import { FormComponent } from "./form.component";
import { SharedState, SHARED_STATE } from "./sharedState.model";
import { Subject } from "rxjs/Subject";

@NgModule({
    imports: [BrowserModule, FormsModule, ModelModule],
    declarations: [TableComponent, FormComponent],
    exports: [ModelModule, TableComponent, FormComponent],
    providers: [{ provide: SHARED_STATE, useValue: new Subject<SharedState>() }]
})
export class CoreModule { }
```

Провайдер приказывает Angular использовать объект `Subject<SharedState>` для разрешения зависимостей маркера `SHARED_STATE`, который предоставляет компоненты с необходимой функциональностью.

В результате преобразование общей службы в `Subject` позволяет компоненту таблицы выдавать события, которые получает компонент формы и которые используются для обновления его состояния без использования неудобного и высокозатратного метода `ngDoCheck`. Также нет необходимости разбираться в том, какие изменения были сгенерированы локальным компонентом, а какие пришли из другого источника, потому что компонент, подписавшийся на `Observable`, знает, что все полученные им события происходят от `Observer`. А это означает, что такие раздражающие проблемы, как невозможность дважды отредактировать один товар, попросту исчезнут (рис. 23.4).

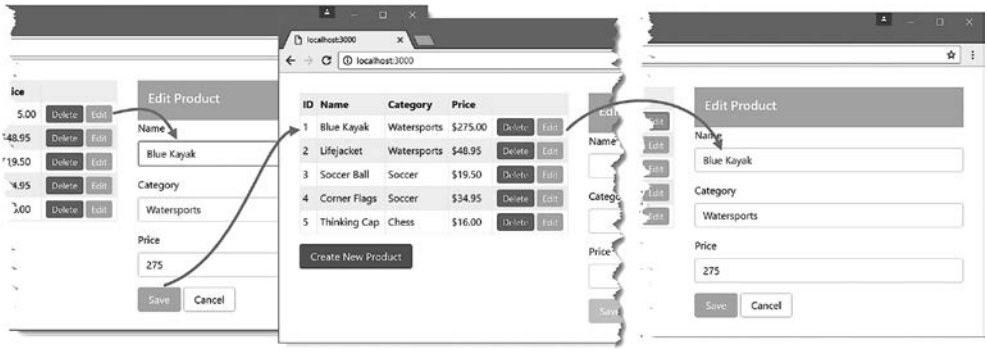


Рис. 23.4. Эффект использования Reactive Extensions

РАЗНОВИДНОСТИ SUBJECT

В листинге 23.5 используется класс Subject — простейший способ создания объекта, который одновременно является Observer и Observable. Главное ограничение такого решения заключается в том, что при создании нового подписчика методом subscribe он не получит событие до следующего вызова метода next. Это может быть неудобно, если экземпляры компонентов или директив создаются динамически и вы хотите, чтобы при создании они получали контекстные данные.

Библиотека Reactive Extensions включает специализированные реализации класса Subject, которые могут использоваться для обходного решения этой проблемы. Класс BehaviorSubject отслеживает последнее обработанное событие и отправляет его новым подписчикам сразу же при вызове метода subscribe. Класс ReplaySubject служит похожим целям, не считая того, что он отслеживает все свои события и отправляет их новым подписчикам, позволяя им «догнать» все события, отправленные перед подпиской.

Использование канала async

Angular включает канал `async`, который может использоваться для потребления объектов `Observable` прямо в представлении, с выбором последнего полученного объекта из последовательности событий. Этот канал является нечистым (см.

главу 18), потому что его изменения инициируются за пределами представления; это означает, что его метод `transform` будет вызываться часто, даже если новое событие не было получено от `Observable`. В листинге 23.6 продемонстрировано добавление канала `async` в представление, находящееся под управлением компонента формы.

Листинг 23.6. Использование канала async в файле form.component.html

```
<div class="bg-primary p-a-1" [class.bg-warning]="editing">
    <h5>{{editing  ? "Edit" : "Create"}} Product</h5>
    Last Event: {{ stateEvents | async | json }}
</div>

<form novalidate #form="ngForm" (ngSubmit)="submitForm(form)" (reset)="resetForm()"
>

    ...элементы опущены для краткости...

</form>
```

Выражение привязки со строковой интерполяцией получает свойство `stateEvents` от компонента (объект `Observable<SharedState>`) и передает его каналу `async`, который отслеживает последнее полученное событие. Фильтр `async` затем передает событие каналу `json`, который создает представление объекта события в формате JSON. В результате вы получаете возможность отслеживать события, полученные компонентом формы (рис. 23.5).

Рис. 23.5. Вывод наблюдаемых событий

Это не самый полезный вариант отображения данных, но он дает информацию, которая может пригодиться при отладке. В данном случае у последнего события значение `mode` равно 1, что соответствует режиму `Edit`, а значение `id` равно 4 (идентификатор товара `Corner Flags`).

Использование канала async с нестандартными каналами

Канал async может использоваться с нестандартными каналами для представления данных события способом, более удобным для пользователя. Для демонстрации создайте файл с именем state.pipe.ts в папке exampleApp/app/core и включите в него определение канала из листинга 23.7.

Листинг 23.7. Содержимое файла state.pipe.ts в папке exampleApp/app/core

```
import { Pipe } from "@angular/core";
import { SharedState, MODES } from "./sharedState.model";
import { Model } from "../model/repository.model";

@Pipe({
    name: "formatState",
    pure: true
})
export class StatePipe {

    constructor(private model: Model) { }

    transform(value: any): string {
        if (value instanceof SharedState) {
            let state = value as SharedState;
            return MODES[state.mode] + (state.id != undefined
                ? ` ${this.model.getProduct(state.id).name}` : "");
        } else {
            return "<No Data>"
        }
    }
}
```

В листинге 23.8 канал добавляется в набор объявлений базового модуля.

ПРИМЕЧАНИЕ

У перечислений TypeScript предусмотрена полезная возможность для получения имени значения. Таким образом, например, выражение MODES[1] возвращает EDIT — это имя значения перечисления MODES с индексом 1. Канал из листинга 23.7 использует эту возможность для вывода обновленной информации состояния для пользователя.

Листинг 23.8. Регистрация канала в файле core.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule } from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { TableComponent } from "./table.component";
import { FormComponent } from "./form.component";
import { SharedState, SHARED_STATE } from "./sharedState.model";
import { Subject } from "rxjs/Subject";
import { StatePipe } from "./state.pipe";
```

```
@NgModule({
    imports: [BrowserModule, FormsModule, ModelModule],
    declarations: [TableComponent, FormComponent, StatePipe],
    exports: [ModelModule, TableComponent, FormComponent],
    providers: [{ provide: SHARED_STATE, useValue: new Subject<SharedState>() }]
})
export class CoreModule { }
```

В листинге 23.9 продемонстрировано применение нового канала для замены встроенного канала `json` в шаблоне, находящемся под управлением компонента формы.

Листинг 23.9. Применение нестандартного канала в файле form.component.html

```
<div class="bg-primary p-a-1" [class.bg-warning]="editing">
    <h5>{{editing  ? "Edit" : "Create"}} Product</h5>
    Last Event: {{ stateEvents | async | formatState }}
</div>

<form novalidate #form="ngForm" (ngSubmit)="submitForm(form)" (reset)="resetForm()"
>

    ...элементы опущены для краткости...

</form>
```

Этот пример показывает, что события, получаемые от объектов `Observable`, могут обрабатываться и преобразовываться, как и любые другие объекты (рис. 23.6); таким образом, нестандартный канал может строиться на основе базовой функциональности, предоставляемой каналом `async`.

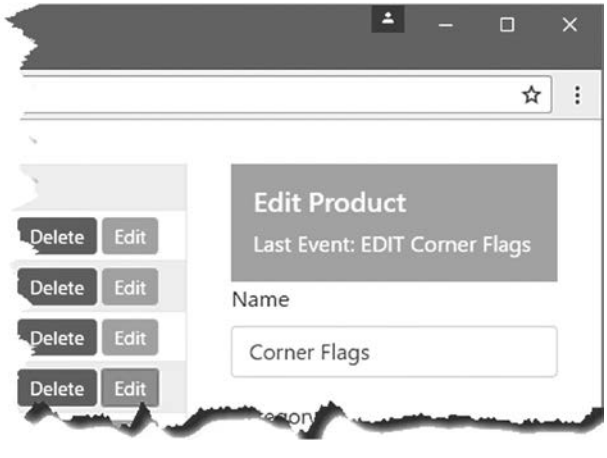


Рис. 23.6. Форматирование значений, полученных через последовательность Observable

Масштабирование функциональных модулей приложения

Структурные блоки Reactive Extensions могут использоваться в разных местах приложения. Это позволяет легко организовать взаимодействие между блоками, даже если использование Reactive Extensions не раскрывается всем взаимодействующим частям приложения. В листинге 23.10 продемонстрировано добавление `Subject` в класс `MessageService` для распространения сообщений, выводимых для пользователя.

Листинг 23.10. Использование Subject в файле message.service.ts

```
import { Injectable } from "@angular/core";
import { Message } from "./message.model";
import { Observable } from "rxjs/Observable";
import { Subject } from "rxjs/Subject";

@Injectable()
export class MessageService {
    private subject = new Subject<Message>();

    reportMessage(msg: Message) {
        this.subject.next(msg);
    }

    get messages(): Observable<Message> {
        return this.subject;
    }
}
```

Предыдущая реализация службы сообщений поддерживала только одного получателя сообщений, которые должны выводиться для пользователя. Я мог бы добавить код для управления несколькими получателями, но с учетом того, что приложение уже использует Reactive Extensions, будет намного проще поручить эту работу классу `Subject`; такое решение хорошо масштабируется и не требует дополнительного кода или тестирования при наличии нескольких подписчиков в приложении.

В листинге 23.11 приведены изменения в компоненте сообщений, который выводит последнее сообщение для пользователя.

Листинг 23.11. Отслеживание сообщений в файле message.component.ts

```
import { Component } from "@angular/core";
import { MessageService } from "./message.service";
import { Message } from "./message.model";
import { Observable } from "rxjs/Observable";

@Component({
    selector: "paMessages",
    moduleId: module.id,
    templateUrl: "message.component.html",
```

```
})
export class MessageComponent {
    lastMessage: Message;

    constructor(messageService: MessageService) {
        messageService.messages.subscribe(m => this.lastMessage = m);
    }
}
```

Остается сгенерировать сообщения, которые бы выводились в приложении. В листинге 23.12 конфигурация базового функционального модуля изменяется так, чтобы провайдер SHARED_STATE применял фабричную функцию для создания объекта Subject, используемого для распространения событий изменения состояния, а также добавлял подписку для передачи событий службе сообщений.

Листинг 23.12. Включение службы сообщений в файл core.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule } from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { TableComponent } from "./table.component";
import { FormComponent } from "./form.component";
import { SharedState, SHARED_STATE } from "./sharedState.model";
import { Subject } from "rxjs/Subject";
import { StatePipe } from "./state.pipe";
import { MessageModule } from "../messages/message.module";
import { MessageService } from "../messages/message.service";
import { Message } from "../messages/message.model";
import { Model } from "../model/repository.model";
import { MODES } from "./sharedState.model";

@NgModule({
    imports: [BrowserModule, FormsModule, ModelModule, MessageModule],
    declarations: [TableComponent, FormComponent, StatePipe],
    exports: [ModelModule, TableComponent, FormComponent],
    providers: [{
        provide: SHARED_STATE,
        deps: [MessageService, Model],
        useFactory: (messageService, model) => {
            let subject = new Subject<SharedState>();
            subject.subscribe(m => messageService.reportMessage(
                    new Message(MODES[m.mode] + (m.id != undefined
                        ? ` ${model.getProduct(m.id).name}` : "")))
                );
            return subject;
        }
    }]
})
export class CoreModule { }
```

Код не самый очевидный, но в результате каждое событие изменения состояния, отправленное компонентом таблицы, будет выводиться компонентом сообщений (рис. 23.7). Reactive Extensions позволяет легко связать части приложения, а код

в листинге выглядит так потому, что он также использует службу модели для получения имен из модели данных (чтобы события лучше читались).

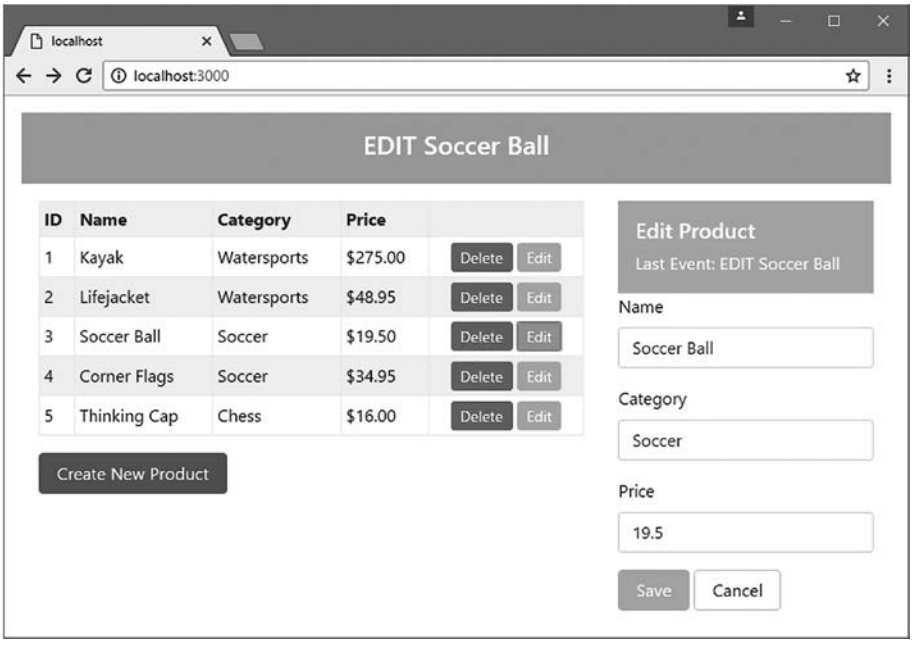


Рис. 23.7. Использование Reactive Extensions в службе сообщений

Расширенные возможности

Примеры в предыдущих разделах демонстрируют простейшее использование `Observable`, `Observer` и `Subject`. Однако при работе с Reactive Extensions доступно значительно больше функциональности, которая может использоваться в сложных или нетривиальных приложениях. Описание всего набора операций можно найти по адресу *http://github.com/Reactive-Extensions/RxJS*, но в этой главе я продемонстрирую небольшое количество возможностей, которые с большой вероятностью понадобятся вам в приложениях Angular (табл. 23.5). Методы, описанные в таблице, предназначены для управления получением событий от объекта `Observable`.

Таблица 23.5. Полезные методы Reactive Extensions для выбора событий

Имя	Описание
filter	Метод вызывает функцию для каждого события, полученного от Observable, и отбрасывает события, для которых функция возвращает false
map	Метод вызывает функцию для преобразования каждого события, полученного от Observable, и передает объект, возвращенный функцией

Имя	Описание
distinctUntilChanged	Метод подавляет события до изменения объекта события
skipWhile	Метод фильтрует события до выполнения заданного условия, после чего все события передаются подписчику
takeWhile	Метод передает события подписчику до выполнения заданного условия, после чего все события отфильтровываются

Фильтрация событий

Метод `filter` получает метод, который проверяет каждый объект, полученный от `Observable`, и выбирает только те объекты, которые подходят по некоторому критерию. Листинг 23.13 демонстрирует использование метода `filter` для отфильтровывания событий, относящихся к конкретному продукту.

Листинг 23.13. Фильтрация событий в файле form.component.ts

```
import { Component, Inject } from "@angular/core";
import { NgForm } from "@angular/forms";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
import { MODES, SharedState, SHARED_STATE } from "./sharedState.model";
import { Observable } from "rxjs/Observable";
import "rxjs/add/operator/filter";

@Component({
    selector: "paForm",
    moduleId: module.id,
    templateUrl: "form.component.html",
    styleUrls: ["form.component.css"]
})
export class FormComponent {
    product: Product = new Product();

    constructor(private model: Model,
        @Inject(SHARED_STATE) private stateEvents: Observable<SharedState>) {

        stateEvents
            .filter(state => state.id != 3)
            .subscribe((update) => {
                this.product = new Product();
                if (update.id != undefined) {
                    Object.assign(this.product, this.model.getProduct(update.id));
                }
                this.editing = update.mode == MODES.EDIT;
            });
    }

    editing: boolean = false;
```

```
    submitForm(form: NgForm) {
        if (form.valid) {
            this.model.saveProduct(this.product);
            this.product = new Product();
            form.reset();
        }
    }

    resetForm() {
        this.product = new Product();
    }
}
```

Чтобы использовать методы из табл. 23.5, необходимо включить команду `import` для соответствующего файла из пакета `rxjs`:

```
...
import "rxjs/add/operator/filter";
...
```

Каталог node_modules/add/operator предоставляет определения этих методов и включает информацию типов для их использования TypeScript. Компилятор TypeScript всегда обращается к папке node_modules при попытке разрешить ссылку, и она исключается из команды `import`.

В нашем примере файл rxjs/add/operator/filter.ts расширяет определение `Observable` и добавляет метод с именем `filter`.

Щелкните на кнопке Edit для товара `Soccer Ball`, идентификатор которого соответствует тому, что ищет функция `filter`. Канал `async` показывает, что событие `EDIT` было отправлено через общую службу, но из-за метода `filter` он не был получен функцией `subscribe` компонента.

В результате форма не отражает изменения состояния и не заполняется данными выбранного товара (рис. 23.8).

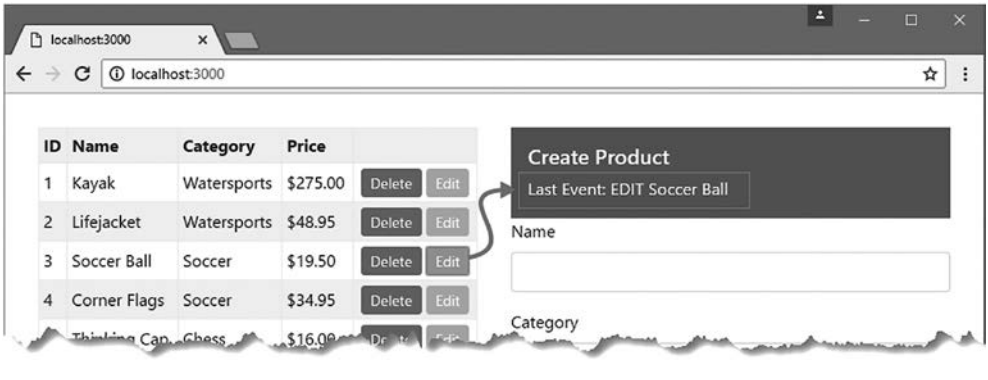


Рис. 23.8. Фильтрация событий

Преобразование событий

Метод `map` предназначен для преобразования объектов, полученных от `Observable`. Вы можете использовать его для произвольного преобразования объектов события, а результат вызова функции заменяет объект события. В листинге 23.14 метод `map` используется для изменения значения свойства объекта события.

Листинг 23.14. Преобразование событий в файле form.component.ts

```
import { Component, Inject } from "@angular/core";
import { NgForm } from "@angular/forms";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
import { MODES, SharedState, SHARED_STATE } from "./sharedState.model";
import { Observable } from "rxjs/Observable";
import "rxjs/add/operator/filter";
import "rxjs/add/operator/map";

@Component({
    selector: "paForm",
    moduleId: module.id,
    templateUrl: "form.component.html",
    styleUrls: ["form.component.css"]
})
export class FormComponent {
    product: Product = new Product();

    constructor(private model: Model,
        @Inject(SHARED_STATE) private stateEvents: Observable<SharedState>) {

        stateEvents
            .map(state => new SharedState(state.mode, state.id == 5 ? 1 : state.id))
            .filter(state => state.id != 3)
            .subscribe((update) => {
                this.product = new Product();
                if (update.id != undefined) {
                    Object.assign(this.product, this.model.getProduct(update.id));
                }
                this.editing = update.mode == MODES.EDIT;
            });
    }

    editing: boolean = false;

    submitForm(form: NgForm) {
        if (form.valid) {
            this.model.saveProduct(this.product);
            this.product = new Product();
            form.reset();
        }
    }

    resetForm() {
        this.product = new Product();
    }
}
```

Функция, передаваемая методу `map` в этом примере, ищет объекты `SharedState` со значением `id`, равным 5, и заменяет его на 1. В результате при щелчке на кнопке Edit у товара `Thinking Cap` для редактирования выбирается товар `Kayak` (рис. 23.9).

ВНИМАНИЕ

При использовании метода map не изменяйте объект, полученный в аргументе функции. Объект передается всем подписчикам по порядку, и все вносимые изменения влияют на последующих подписчиков. Это означает, что одни подписчики получают исходный объект, а другие — объект, возвращенный функцией map. Вместо этого создайте новый объект, как показано в листинге 23.14.

Следует заметить, что методы, используемые для подготовки и создания подписки на объект `Observable`, могут объединяться в цепочку. В нашем примере результат метода `map` используется в качестве входных данных метода `filter`, результат которого затем передается функции метода `subscribe`. Такое объединение методов позволяет создавать сложные правила для обработки и получения событий.

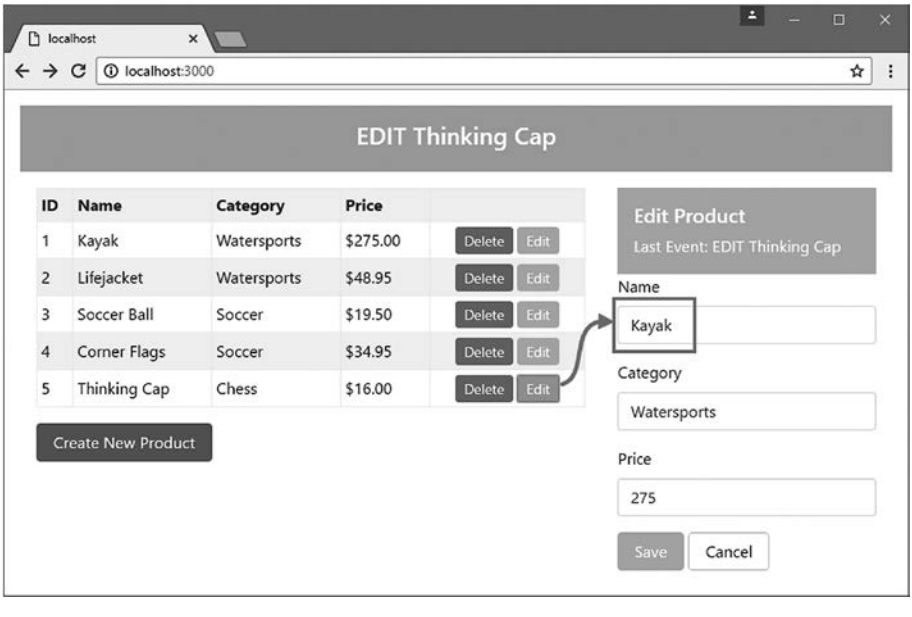


Рис. 23.9. Преобразование событий

Использование разных объектов событий

Метод `map` может использоваться для создания произвольных объектов и не ограничивается изменением значений свойств получаемых объектов. В листинге 23.15 метод `map` используется для получения числа, в значении которого кодируется операция и объект, к которому он применяется.

Листинг 23.15. Проецирование другого типа в файле form.component.ts

```
...
constructor(private model: Model,
    @Inject(SHARED_STATE) private stateEvents: Observable<SharedState>) {

    stateEvents
        .map(state => state.mode == MODES.EDIT ? state.id : -1)
        .filter(id => id != 3)
        .subscribe((id) => {
            this.editing = id != -1;
            this.product = new Product();
            if (id != -1) {
                Object.assign(this.product, this.model.getProduct(id))
            }
        });
}
...
```

Представление как операции, так и ее объекта в простом типе данных не дает никаких преимуществ. Обычно оно только создает проблемы, потому что компонент предполагает, что в модели никогда не будет объекта со свойством `id`, равным –1. Но в нашем простом примере оно демонстрирует, как метод `map` может проецировать разные типы и как эти типы затем передаются по цепочке методов Reactive Extensions; это означает, что значения `number`, генерируемые методом `map`, передаются для обработки методу `filter`, а затем методу `subscribe`, функции которых были обновлены для работы с новыми значениями данных.

Получение уникальных событий

Метод `distinctUntilChanged` фильтрует последовательность событий, чтобы подписчику передавались только уникальные значения. Чтобы понять, какие проблемы могут решаться таким образом, щелкните на кнопке Edit для товара `Kayak` и измените значение в поле Category. Не щелкая на кнопке Save, снова щелкните на кнопке Edit товара `Kayak`; вы увидите, что внесенные изменения пропадают. В листинге 23.16 метод `distinctUntilChanged` добавляется в цепочку методов, чтобы он применялся только к значениям `number`, производимым методом `map`. Методам `filter` и `subscribe` будут передаваться только различающиеся значения.

Листинг 23.16. Исключение дубликатов событий в файле form.component.ts

```
import { Component, Inject } from "@angular/core";
import { NgForm } from "@angular/forms";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
import { MODES, SharedState, SHARED_STATE } from "./sharedState.model";
import { Observable } from "rxjs/Observable";
import "rxjs/add/operator/filter";
import "rxjs/add/operator/map";
import "rxjs/add/operator/distinctUntilChanged";
```

```
@Component({
    selector: "paForm",
    moduleId: module.id,
    templateUrl: "form.component.html",
    styleUrls: ["form.component.css"]
})
export class FormComponent {
    product: Product = new Product();

    constructor(private model: Model,
        @Inject(SHARED_STATE) private stateEvents: Observable<SharedState>) {

        stateEvents
            .map(state => state.mode == MODES.EDIT ? state.id : -1)
            .distinctUntilChanged()
            .filter(id => id != 3)
            .subscribe((id) => {
                this.editing = id != -1;
                this.product = new Product();
                if (id != -1) {
                    Object.assign(this.product, this.model.getProduct(id))
                }
            });
    }

    editing: boolean = false;

    submitForm(form: NgForm) {
        if (form.valid) {
            this.model.saveProduct(this.product);
            this.product = new Product();
            form.reset();
        }
    }

    resetForm() {
        this.product = new Product();
    }
}
```

Повторив процесс редактирования товара `Kayak`, вы увидите, что изменения уже не теряются при щелчке на кнопке Edit для редактируемого товара, потому что она сгенерирует такое же значение, как для предыдущего события. Редактирование другого товара приведет к тому, что метод `map` сгенерирует другое значение `number`, которое будет передано методу `distinctUntilChanged`.

Нестандартная проверка равенства

Метод `distinctUntilChanged` способен выполнять простые сравнения для простых типов данных (таких, как `number`), но он не умеет сравнивать объекты и счи-

тает, что любые два объекта различны. Для решения этой проблемы можно задать функцию сравнения, которая будет использоваться для проверки событий на предмет различий (листинг 23.17).

Листинг 23.17. Проверка равенства в файле form.component.ts

```
...
constructor(private model: Model,
    @Inject(SHARED_STATE) private stateEvents: Observable<SharedState>) {

    stateEvents
        .distinctUntilChanged((firstState, secondState) =>
            firstState.mode == secondState.mode && firstState.id == secondState.id)
        .subscribe(update => {
            this.product = new Product();
            if (update.id != undefined) {
                Object.assign(this.product, this.model.getProduct(update.id));
            }
            this.editing = update.mode == MODES.EDIT;
        });
}
...
```

В листинге удаляются методы `map` и `filter`, а методу `distinctUntilChanged` передается функция, которая сравнивает объекты `SharedState`, сопоставляя их свойства `mode` и `id`. Различающиеся объекты передаются функции, предоставленной методу `subscribe`.

Передача и игнорирование событий

Методы `skipWhile` и `takeWhile` используются для определения условий, по которым события отфильтровываются или передаются подписчику. При использовании этих методов необходима осторожность — слишком легко задать условия, которые намертво заблокируют получение событий подписчиком. В листинге 23.18 метод `skipWhile` используется для фильтрации событий до того момента, когда пользователь щелкнет на кнопке Create New Product, после чего события снова будут передаваться.

Листинг 23.18. Игнорирование событий в файле form.component.ts

```
import { Component, Inject } from "@angular/core";
import { NgForm } from "@angular/forms";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
import { MODES, SharedState, SHARED_STATE } from "./sharedState.model";
import { Observable } from "rxjs/Observable";
import "rxjs/add/operator/filter";
import "rxjs/add/operator/map";
import "rxjs/add/operator/distinctUntilChanged";
import "rxjs/add/operator/skipWhile";
```

```
@Component({
    selector: "paForm",
    moduleId: module.id,
    templateUrl: "form.component.html",
    styleUrls: ["form.component.css"]
})
export class FormComponent {
    product: Product = new Product();

    constructor(private model: Model,
        @Inject(SHARED_STATE) private stateEvents: Observable<SharedState>) {

        stateEvents
            .skipWhile(state => state.mode == MODES.EDIT)
            .distinctUntilChanged((firstState, secondState) =>
                firstState.mode == secondState.mode
                    && firstState.id == secondState.id)
            .subscribe(update => {
                this.product = new Product();
                if (update.id != undefined) {
                    Object.assign(this.product, this.model.getProduct(update.id));
                }
                this.editing = update.mode == MODES.EDIT;
            });
    }

    editing: boolean = false;
    submitForm(form: NgForm) {
        if (form.valid) {
            this.model.saveProduct(this.product);
            this.product = new Product();
            form.reset();
        }
    }

    resetForm() {
        this.product = new Product();
    }
}
```

При щелчке на кнопках Edit в таблице по-прежнему будут генерироваться события, которые будут отображаться каналом `async`, подписанным на `Subject` без фильтрации или подавления. Но компонент формы не получает эти события (рис. 23.10), потому что его подписка отфильтровывается методом `skipWhile`, пока не будет получено событие, свойство `mode` которого отлично от `MODES.EDIT`. Щелчок на кнопке Create New Product генерирует событие, которое завершает режим пропуска, после чего компонент будет получать все события.

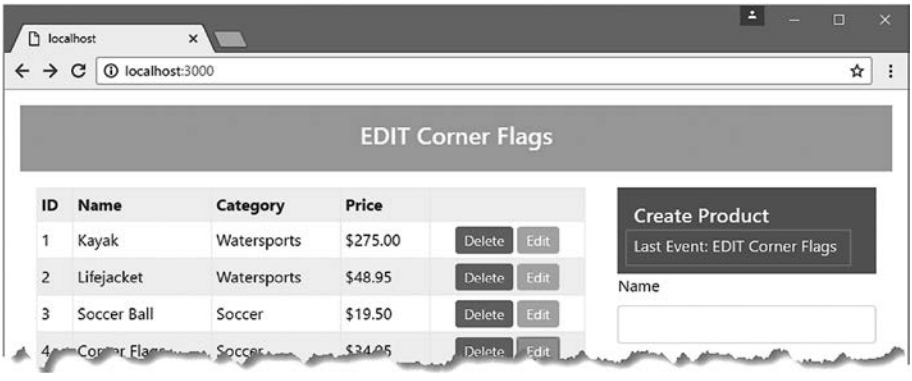


Рис. 23.10. Игнорирование событий

Итоги

В этой главе я представил пакет Reactive Extensions и объяснил, как использовать его для обработки изменений в тех частях приложения, которыми не управляет процесс обнаружения изменений Angular. Вы узнали, как объекты `Observable`, `Observer` и `Subject` используются для распространения событий в приложениях, как работает встроенный канал `async`, и познакомились с самыми полезными операторами, предоставляемыми библиотекой Reactive Extensions для управления потоком событий для подписчика. В следующей главе я объясню, как создавать асинхронные запросы HTTP в приложениях Angular и как использовать REST-совместимые веб-службы.

24 Асинхронные запросы HTTP

Начиная с главы 11 во всех примерах использовались статические данные, жестко запрограммированные в приложении. В этой главе я покажу, как использовать асинхронные запросы HTTP (часто называемые *запросами Ajax*) для взаимодействия с веб-службами для получения реальных данных. В табл. 24.1 запросы HTTP представлены в контексте.

Таблица 24.1. Асинхронные запросы HTTP в контексте

Вопрос	Ответ
Что это такое?	Асинхронные запросы HTTP отправляются браузером по требованию приложения. Термин «асинхронный» означает, что приложение продолжает работать, пока браузер ожидает ответа сервера
Для чего они нужны?	Асинхронные запросы HTTP позволяют приложениям Angular взаимодействовать с веб-службами, чтобы хранимые данные можно было загружать в приложении, а изменения можно было отправлять на сервер для сохранения
Как они используются?	Запросы создаются с использованием класса Http, который предоставляется как служба через механизм внедрения зависимостей. Этот класс предоставляет Angular-совместимую обертку для браузерной функциоальности XMLHttpRequest
Есть ли у них недостатки или скрытые проблемы?	Функциональность HTTP требует использования объектов Observable библиотеки Reactive Extensions (см. главу 23)
Есть ли альтернативы?	При желании можно работать напрямую с браузерным объектом XMLHttpRequest, а некоторые приложения (те, которым не нужно работать с хранимыми данными) пишутся вообще без запросов HTTP

ПРОБЛЕМЫ С СЕРВЕРАМИ HTTP

В первом издании книги я выбрал пакет Deployd для работы с REST-совместимой веб-службой. Как выяснилось, решение было неудачным, потому что оно некорректно работало с некоторыми средствами чтения. Из-за этого мне пришлось обновить главы, в которых используется parse.com — коммерческая платформа хостинга, принадлежащая Facebook. Я думал, что это решение станет платформой для долгосрочного использования, но компания Facebook недавно объявила о закрытии parse.com.

В этом издании я выбрал превосходный пакет json-server и надеюсь, что на этот раз проблем не будет — при условии, что вы используете версии пакета из листинга 24.1.

Если вы столкнетесь с проблемами, загляните на сайт apress.com и посмотрите, нет ли на нем обновленных версий глав (вроде тех, которые я написал для предыдущего издания). Если таких материалов нет или вам понадобится дополнительная помощь, напишите мне по адресу adam@adam-freeman.com — я постараюсь помочь.

В табл. 24.2 приведена краткая сводка материала главы.

Таблица 24.2. Сводка материала главы

Проблема	Решение	Листинг
Отправка запросов HTTP в приложениях Angular	Используйте службу Http	1–8
Выполнение REST-операций	Используйте метод HTTP и URL для задания операции и цели этой операции	9–11
Создание междоменных запросов (CORS)	Служба Http поддерживает CORS автоматически, но также поддерживаются запросы JSONP	12–13
Включение заголовков в запрос	Задайте свойство headers в объекте Request	14–15
Реакция на ошибку HTTP	Создайте класс обработчика ошибки	16–19

Подготовка проекта

Функциональность, описанная в этой главе, зависит от модуля HTTP, который необходимо добавить в приложение. В листинге 24.1 модуль добавляется в список пакетов.

Листинг 24.1. Добавление пакетов и сценария в файл package.json

```
{
  "dependencies": {
    "@angular/common": "2.2.0",
    "@angular/compiler": "2.2.0",
    "@angular/core": "2.2.0",
    "@angular/platform-browser": "2.2.0",
    "@angular/platform-browser-dynamic": "2.2.0",
    "@angular/upgrade": "2.2.0",
    "@angular/forms": "2.2.0",
    "@angular/http": "2.2.0",
    "reflect-metadata": "0.1.8",
    "rxjs": "5.0.0-beta.12",
    "zone.js": "0.6.26",
    "core-js": "2.4.1",
    "classlist.js": "1.1.20150312",
    "systemjs": "0.19.40",
    "bootstrap": "4.0.0-alpha.4",
    "intl": "1.2.4"
  },

  "devDependencies": {
    "lite-server": "2.2.2",
    "typescript": "2.0.2",
    "typings": "1.3.2",
    "concurrently": "2.2.0",
    "systemjs-builder": "0.15.32",
    "json-server": "0.8.21"
  },

  "scripts": {
    "start": "concurrently \"npm run tscwatch\" \"npm run lite\" \"npm run json\" ",
    "tsc": "tsc",
    "tscwatch": "tsc -w",
    "lite": "lite-server",
    "json": "json-server --p 3500 restData.js",
    "typings": "typings",
    "postinstall": "typings install"
  }
}
```

Модуль JavaScript с именем `@angular/http` предоставляет функциональность, необходимую для этой главы; он добавляется в секцию `dependencies` файла. Кроме того, файл package.json добавляет в проект пакет `json-server` и сценарий для его запуска. Пакет упрощает создание веб-служб; он будет использован для создания внутренней подсистемы, к которой примеры этой главы смогут отправлять запросы HTTP.

Настройка загрузчика модулей JavaScript

Следующим шагом станет обновление файла конфигурации SystemJS, чтобы он мог разрешать зависимости модуля форм (листинг 24.2).

Листинг 24.2. Добавление модуля HTTP в файл systemjs.config.js

```
(function (global) {

    var paths = {
        "@angular/*": "node_modules/@angular/*"
    }

    var packages = { "app": {} };
    var angularModules = ["common", "compiler",
     "core", "platform-browser", "platform-browser-dynamic", "forms", "http"];

    angularModules.forEach(function (pkg) {
        packages["@angular/" + pkg] = {
            main: "/bundles/" + pkg + ".umd.min.js"
        };
    });

    System.config({ paths: paths, packages: packages });

})(this);
```

Настройка функционального модуля модели

JavaScript-модуль `@angular/http` содержит модуль Angular с именем `HttpModule`, который необходимо импортировать в приложение в корневом модуле или в одном из функциональных модулей. Функциональность HTTP понадобится только в модели данных, поэтому в листинге 24.3 приведены изменения в файле model.module.ts из папки exampleApp/app/model.

Листинг 24.3. Импортирование модуля в файле model.module.ts

```
import { NgModule } from "@angular/core";
import { HttpModule } from "@angular/http"
import { StaticDataSource } from "./static.datasource";
import { Model } from "./repository.model";

@NgModule({
    imports: [HttpModule],
    providers: [Model, StaticDataSource]
})
export class ModelModule { }
```

Запись `json` в секции `scripts` из листинга 24.1 сообщает, что пакет `json-server` должен вести прослушивание запросов HTTP на порте 3500, а данные берутся из файла restData.js. Чтобы предоставить сами данные, создайте файл restData.js в папке exampleApp и включите в него код из листинга 24.4.

Листинг 24.4. Содержимое файла restData.js в папке exampleApp

```
module.exports = function () {
    var data = {
        products: [
            { id: 1, name: "Kayak", category: "Watersports", price: 275 },
            { id: 2, name: "Lifejacket", category: "Watersports", price: 48.95 },
            { id: 3, name: "Soccer Ball", category: "Soccer", price: 19.50 },
            { id: 4, name: "Corner Flags", category: "Soccer", price: 34.95 },
            { id: 5, name: "Stadium", category: "Soccer", price: 79500 },
            { id: 6, name: "Thinking Cap", category: "Chess", price: 16 },
            { id: 7, name: "Unsteady Chair", category: "Chess", price: 29.95 },
            { id: 8, name: "Human Chess Board", category: "Chess", price: 75 },
            { id: 9, name: "Bling Bling King", category: "Chess", price: 1200 }
        ]
    }
    return data
}
```

Пакет `json-server` может работать с файлами JavaScript или JSON. Если вы используете файл JSON, его содержимое будет изменяться в соответствии с изменениями, внесенными клиентами. Я выбрал вариант с JavaScript, который позволяет генерировать данные на программном уровне; это означает, что при перезапуске процесса произойдет возврат к исходным данным.

Обновление компонента формы

В главе 23 компонент, управляющий формой HTML, был настроен на игнорирование событий, генерируемых компонентом таблицы, до первого щелчка на кнопке Create New Product. Чтобы предотвратить странные результаты, листинг 24.5 блокирует методы `skipWhile` и `distinctUntilChanged`, которые были применены к `Observable`.

Листинг 24.5. Отключение игнорирования событий в файле form.component.ts

```
...
constructor(private model: Model,
    @Inject(SHARED_STATE) private stateEvents: Observable<SharedState>) {

    stateEvents
        //.skipWhile(state => state.mode == MODES.EDIT)
        //.distinctUntilChanged((firstState, secondState) =>
        //    firstState.mode == secondState.mode
        //    && firstState.id == secondState.id)
        .subscribe(update => {
            this.product = new Product();
            if (update.id != undefined) {
```

```
                Object.assign(this.product, this.model.getProduct(update.id));
            }
            this.editing = update.mode == MODES.EDIT;
        });
}
...
```

Запуск проекта

Сохраните изменения и выполните следующую команду из папки exampleApp, чтобы загрузить и установить модуль Angular Http, а также пакет `json-server`:

```
npm install
```

После того как обновление будет завершено, выполните следующую команду, чтобы запустить компилятор TypeScript, сервер HTTP для разработки и REST-совместимую веб-службу:

```
npm start
```

Открывается новое окно (или вкладка) браузера с контентом, показанным на рис. 24.1.

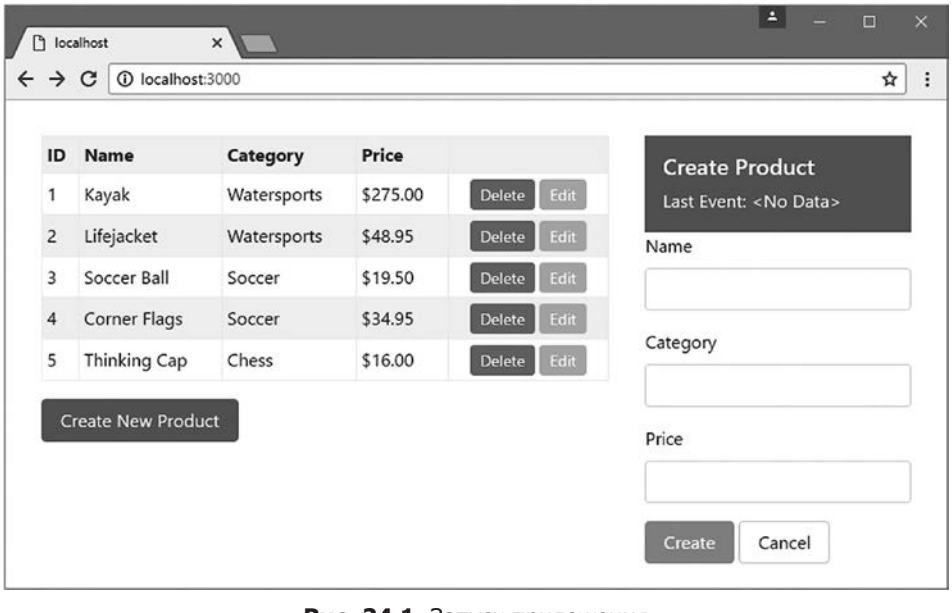


Рис. 24.1. Запуск приложения

Чтобы протестировать REST-совместимые веб-службы, введите следующий URL-адрес в окне браузера:

```
http://localhost:3500/products/2
```

Сервер отвечает следующими данными:

```
{ "id": 2, "name": "Lifejacket", "category": "Watersports", "price": 48.95 }
```

REST-совместимые веб-службы

Самый распространенный механизм поставки данных приложению — использование паттерна *REST* (Representational State Transfer) для создания веб-службы данных. Подробной спецификации REST не существует, что порождает множество разных подходов, объединяемых под знаменем «REST-совместимости». Однако существуют некоторые объединяющие идеи, которые принесут пользу при разработке веб-приложений.

Основная концепция REST-совместимой веб-службы — имитация характеристик HTTP, чтобы методы запросов, также называемые командами (verbs), задавали операцию, выполняемую сервером, а URL-адрес запроса определял один или несколько объектов данных, к которым должна применяться операция.

Например, URL-адрес, ссылающийся на конкретный товар в нашем примере, может выглядеть так:

```
http://localhost:3500/products/2
```

Последний сегмент URL — `products` — задает коллекцию объектов, с которой будет выполняться операция; он позволяет одному серверу предоставить несколько служб, каждая из которых работает с собственными данными. Второй сегмент — `2` — выбирает отдельный объект в коллекции `products`. В данном случае значение свойства `id` однозначно идентифицирует объект и используется в URL (оно определяет объект `Lifejacket`).

Команда (или метод) HTTP, используемая для выдачи запроса, сообщает REST-совместимому серверу, какая операция должна быть выполнена с заданным объектом. Когда вы тестировали REST-совместимый сервер в предыдущем разделе, браузер отправил запрос HTTP Get, который интерпретируется сервером как инструкция на выборку заданного объекта и отправку его клиенту. Именно по этой причине браузер выводит представление объекта `Lifejacket` в формате JSON.

В табл. 24.3 представлены самые распространенные комбинации методов HTTP и URL с объяснениями того, что делает каждая комбинация при отправке REST-совместимому серверу.

Таблица 24.3. Стандартные команды HTTP и их использование в REST-совместимых веб-службах

Команда HTTP	URL	Описание
GET	/products	Комбинация читает все объекты из коллекции products
GET	/products/2	Комбинация читает из коллекции products объект со значением id, равным 2

Команда HTTP	URL	Описание
POST	/products	Комбинация добавляет новый объект в коллекцию products. Тело запроса содержит представление нового объекта в формате JSON
PUT	/products/2	Комбинация заменяет в коллекции products объект со значением id, равным 2. Тело запроса содержит представление заменяющего объекта в формате JSON
PATCH	/products/2	Комбинация используется для обновления подмножества свойств объекта из коллекции products, значение id которого равно 2. Тело запроса содержит представление обновляемых свойств и новых значений в формате JSON
DELETE	/products/2	Комбинация удаляет из коллекции products объект со значением id, равным 2

Будьте внимательны: в работе некоторых веб-служб существуют серьезные различия, обусловленные различиями между фреймворками, использованными для их создания, и личными предпочтениями группы разработки. Обязательно проверьте, как веб-служба использует команды и что требуется от URL и тела запроса для выполнения операций.

На практике часто встречаются веб-службы, не принимающие тело запроса со значением `id` (чтобы его уникальное значение генерировалось хранилищем данных на сервере), и веб-службы, не поддерживающие часть команд (часто игнорируются запросы PATCH и принимаются только обновления, использующие запросы PUT).

Замена статического источника данных

Начать лучше всего с замены статического источника данных другим, получающим данные от REST-совместимой веб-службы. Этот шаг послужит основой для описания поддержки запросов HTTP в Angular и интеграции их в приложение.

Создание новой службы источника данных

Чтобы создать новый источник данных, создайте файл rest.datasource.ts в папке exampleApp/app/model и включите в него команды из листинга 24.6.

Листинг 24.6. Содержимое файла rest.datasource.ts в папке exampleApp/app/model

```
import { Injectable, Inject, OpaqueToken } from "@angular/core";
import { Http } from "@angular/http";
import { Observable } from "rxjs/Observable";
```

```
import { Product } from "./product.model";
import "rxjs/add/operator/map";

export const REST_URL = new OpaqueToken("rest_url");

@Injectable()
export class RestDataSource {

    constructor(private http: Http,
        @Inject(REST_URL) private url: string) { }

    getData(): Observable<Product[]> {
        return this.http.get(this.url).map(response => response.json());
    }
}
```

Этот класс кажется простым, но в нем задействованы важные механизмы, которые будут описаны ниже.

Создание запроса HTTP

Для создания асинхронных запросов HTTP Angular предоставляет класс `Http`, который определяется в модуле `@angular/http` и предоставляется как служба в функциональном модуле `HttpModule`. Источник данных объявляет зависимость от класса `Http` через конструктор:

```
...
constructor(private http: Http, @Inject(REST_URL) private url: string) { }
...
```

Другой аргумент конструктора используется для того, чтобы URL, по которому отправляются запросы, не нужно было жестко кодировать в источнике данных. Я создам провайдер с использованием маркера `REST_URL` при настройке функционального модуля. Объект `Http`, полученный через конструктор, используется для создания запросов HTTP `GET` в методе `getData` источника данных:

```
...
getData(): Observable<Product[]> {
    return this.http.get(this.url).map(response => response.json());
}
...
```

Класс `Http` определяет набор методов для создания запросов HTTP с использованием разных команд HTTP (табл. 24.4).

ПРИМЕЧАНИЕ

Методы из табл. 24.4 могут получать необязательный объект конфигурации (см. раздел «Настройка заголовков запроса»).

Таблица 24.4. Методы HTTP

Имя	Описание
get(url)	Метод отправляет запрос GET по заданному URL-адресу
post(url, body)	Метод отправляет запрос POST, используя заданный объект как тело запроса
put(url, body)	Метод отправляет запрос PUT, используя заданный объект как тело запроса
patch(url, body)	Метод отправляет запрос PATCH, используя заданный объект как тело запроса
delete(url)	Метод отправляет запрос DELETE по заданному URL-адресу
head(url)	Метод отправляет запрос HEAD, который работает так же, как и запрос GET, за исключением того, что сервер возвращает только заголовки без тела запроса
options(url)	Метод отправляет запрос OPTIONS по заданному URL-адресу
request(request)	Метод может использоваться для отправки запроса с любой командой (см. раздел «Консолидация запросов HTTP»)

Обработка ответа

Методы из табл. 24.4 возвращают объект Reactive Extensions `Observable<Response[]>` (см. главу 23), который отправляет событие при получении ответа от сервера.

Класс `Response`, который также определяется в модуле JavaScript `@angular/http`, используется для представления ответа от сервера. В табл. 24.5 описаны важнейшие методы и свойства, определяемые классом `Response`.

Таблица 24.5. Полезные методы и свойства Response

Имя	Описание
ok	Свойство типа boolean возвращает true, если код статуса ответа лежит в диапазоне от 200 до 299 (признак успешного запроса)
status	Свойство типа number возвращает код статуса из ответа
statusText	Свойство типа string возвращает сообщение с описанием кода статуса из ответа
url	Свойство типа string возвращает запрошенный URL-адрес
totalBytes	Свойство типа number возвращает ожидаемый размер ответа
headers	Свойство возвращает объект Headers, который предоставляет доступ к заголовкам ответа. За подробной информацией о классе Headers обращайтесь к разделу «Настройка заголовков запроса»
json()	Метод интерпретирует данные ответа как формат JSON и пытается разобрать их для создания объектов JavaScript
text()	Метод возвращает данные ответа в виде данных string
blob()	Метод возвращает данные ответа в виде двоичного объекта
arrayBuffer()	Метод возвращает данные ответа в виде массива

REST-совместимый веб-сервер возвращает данные в формате JSON, который стал фактическим стандартом для веб-служб, а метод `json` разбирает ответ и создает массив объектов JavaScript.

```
...
getData(): Observable<Product[]> {
    return this.http.get(this.url).map(response => response.json());
}
...
```

В результате объект `Observable<Response>`, возвращенный методом `Http.get`, преобразуется в `Observable<Product[]>` комбинацией методов `Observable.map` и `Response.json`, а подписчикам `Observer` отправляется массив объектов.

ВНИМАНИЕ

Методы из табл. 24.4 подготавливают запрос HTTP, но он не отправляется серверу до вызова метода subscribe объекта Observer. Впрочем, будьте внимательны: запрос отправляется при каждом вызове subscribe, и вы можете непреднамеренно отправить один запрос несколько раз.

Настройка источника данных

Затем необходимо настроить провайдер для нового источника данных и создать провайдера для настройки его URL-адресом, по которому будут отправляться запросы. В листинге 24.7 показаны изменения в файле model.module.ts.

Листинг 24.7. Настройка источника данных в файле model.module.ts

```
import { NgModule } from "@angular/core";
import { HttpModule } from "@angular/http"
//import { StaticDataSource } from "./static.datasource";
import { Model } from "./repository.model";
import { RestDataSource, REST_URL } from "./rest.datasource";

@NgModule({
    imports: [HttpModule],
    providers: [Model, RestDataSource,
        { provide: REST_URL, useValue: `http://${location.hostname}:3500/products`
}]
})
export class ModelModule { }
```

Два новых провайдера определяют класс `RestDataSource` как службу и используют маркер `REST_URL` для настройки URL веб-службы. Я убрал провайдер для класса `StaticDataSource`, он больше не нужен.

ИЗОЛЯЦИЯ ИСТОЧНИКА ДАННЫХ ДЛЯ МОДУЛЬНОГО ТЕСТИРОВАНИЯ

Источники данных, выдающие сетевые запросы, трудно изолировать для модульного тестирования, потому что разработчику трудно отличить поведение класса источника данных от неожиданных эффектов, вызванных работой сети и сервера.

Один из пакетов NPM предоставляет службу в памяти, которая может использоваться для моделирования запросов HTTP с локально определенными данными. Этот механизм позволяет изолировать класс источника данных и убедиться в том, что тестируется только его поведение. Этот пакет называется angular-in-memory-web-api, а его описание доступно по адресу https://github.com/angular/in-memory-web-api.

Использование REST-совместимого источника данных

Остается обновить класс репозитория, чтобы он объявлял зависимость от нового источника данных и использовал его для получения данных приложения (листинг 24.8).

Листинг 24.8. Использование нового источника данных в файле repository.model.ts

```
import { Injectable } from "@angular/core";
import { Product } from "./product.model";
//import { StaticDataSource } from "./static.datasource";
import { Observable } from "rxjs/Observable";
import { RestDataSource } from "./rest.datasource";

@Injectable()
export class Model {
    private products: Product[] = new Array<Product>();
    private locator = (p: Product, id: number) => p.id == id;

    constructor(private dataSource: RestDataSource) {
        //this.products = new Array<Product>();
        //this.dataSource.getData().forEach(p => this.products.push(p));
        this.dataSource.getData().subscribe(data => this.products = data);
    }

    getProducts(): Product[] {
        return this.products;
    }

    getProduct(id: number): Product {
        return this.products.find(p => this.locator(p, id));
    }

    saveProduct(product: Product) {
        if (product.id == 0 || product.id == null) {
            product.id = this.generateID();
            this.products.push(product);
        } else {
            let index = this.products
                .findIndex(p => this.locator(p, product.id));
            this.products.splice(index, 1, product);
        }
    }

    deleteProduct(id: number) {
        let index = this.products.findIndex(p => this.locator(p, id));
        if (index > -1) {
```

```
            this.products.splice(index, 1);
        }
    }

    private generateID(): number {
        let candidate = 100;
        while (this.getProduct(candidate) != null) {
            candidate++;
        }
        return candidate;
    }
}
```

Зависимость конструктора изменена таким образом, чтобы репозиторий получал объект `RestDataSource` при создании. В конструкторе вызывается метод `getData` источника данных, а метод `subscribe` используется для получения объектов данных, возвращенных сервером, и их обработки.

После сохранения изменений браузер перезагрузит приложение, в котором будет использоваться новый источник данных. Асинхронный запрос HTTP будет отправлен REST-совместимой веб-службе, которая вернет больший набор объектов данных (рис. 24.2).

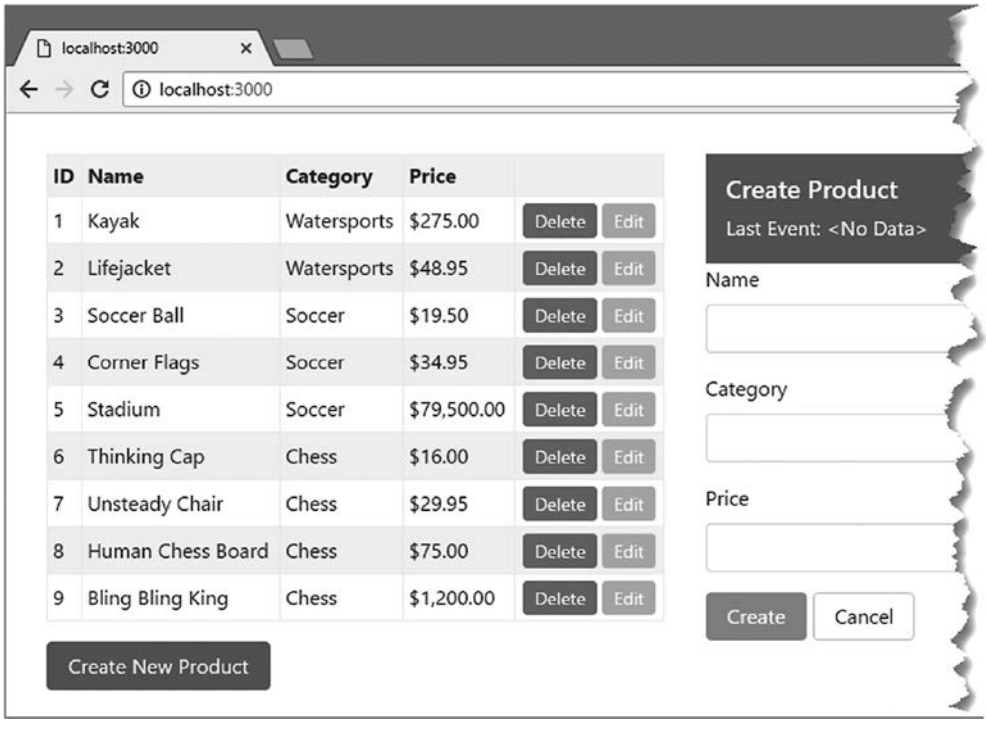


Рис. 24.2. Получение данных приложения

Сохранение и удаление данных

Источник данных может получать данные от сервера, но он также должен передавать данные в другом направлении, чтобы сохранять изменения, внесенные пользователем в объекты модели и вновь созданные объекты. В листинге 24.9 в класс источника данных добавляются методы отправки запросов HTTP для сохранения или обновления объектов с использованием класса Angular `Http`.

Листинг 24.9. Отправка данных в файле rest.datasource.ts

```
import { Injectable, Inject, OpaqueToken } from "@angular/core";
import { Http } from "@angular/http";
import { Observable } from "rxjs/Observable";
import { Product } from "./product.model";
import "rxjs/add/operator/map";

export const REST_URL = new OpaqueToken("rest_url");

@Injectable()
export class RestDataSource {

    constructor(private http: Http,
        @Inject(REST_URL) private url: string) { }

    getData(): Observable<Product[]> {
        return this.http.get(this.url)
            .map(response => response.json());
    }

    saveProduct(product: Product): Observable<Product> {
        return this.http.post(this.url, product)
            .map(response => response.json());
    }

    updateProduct(product: Product): Observable<Product> {
        return this.http.put(`${this.url}/${product.id}`, product)
            .map(response => response.json());
    }

    deleteProduct(id: number): Observable<Product> {
        return this.http.delete(`${this.url}/${id}`)
            .map(response => response.json());
    }
}
```

Методы `saveProduct`, `updateProduct` и `deleteProduct` строятся по одной схеме: они вызывают один из методов класса `Http`, используют метод `map` для обработки объекта `Response`, произведенного запросом HTTP, и возвращают результат `Observable<Product>`. Методы различаются по URL-адресу, на который отправляются запросы.

При сохранении нового объекта идентификатор объекта генерируется сервером, чтобы он был уникальным и клиент не мог случайно использовать один иденти-

фикатор для разных объектов. В данной ситуации используется метод POST, а запрос отправляется на URL `/products`.

При обновлении и удалении существующего объекта идентификатор уже известен и запрос PUT отправляется на URL-адрес, включающий идентификатор. Таким образом, запрос для обновления объекта с идентификатором 2, например, отправляется на URL `/products/2`. Для удаления объекта запрос DELETE будет отправлен по тому же URL.

У всех этих методов есть нечто общее: сервер является авторитетным хранилищем данных, а ответ от сервера содержит официальную версию объекта, который был сохранен сервером. Этот объект возвращается в результатах этих методов, передаваемых через `Observable<Product>`.

В листинге 24.10 показаны соответствующие изменения в классе репозитория, использующие новую функциональность источника данных.

Листинг 24.10. Использование функциональности источника данных в файле repository.model.ts

```
import { Injectable } from "@angular/core";
import { Product } from "./product.model";
import { Observable } from "rxjs/Observable";
import { RestDataSource } from "./rest.datasource";

@Injectable()
export class Model {
    private products: Product[] = new Array<Product>();
    private locator = (p: Product, id: number) => p.id == id;

    constructor(private dataSource: RestDataSource) {
        this.dataSource.getData().subscribe(data => this.products = data);
    }

    getProducts(): Product[] {
        return this.products;
    }

    getProduct(id: number): Product {
        return this.products.find(p => this.locator(p, id));
    }

    saveProduct(product: Product) {
        if (product.id == 0 || product.id == null) {
            this.dataSource.saveProduct(product)
                .subscribe(p => this.products.push(p));
        } else {
            this.dataSource.updateProduct(product).subscribe(p => {
                let index = this.products
                    .findIndex(item => this.locator(item, p.id));
                this.products.splice(index, 1, p);
            });
        }
    }
```

```
    deleteProduct(id: number) {
        this.dataSource.deleteProduct(id).subscribe(() => {
            let index = this.products.findIndex(p => this.locator(p, id));
            if (index > -1) {
                this.products.splice(index, 1);
            }
        });
    }
}
```

Эти изменения используют источник данных для отправки обновлений серверу, а результаты — для обновления локально хранимых данных, чтобы они выводились в других частях приложения. Чтобы протестировать изменения, щелкните на кнопке Edit товара `Kayak` и измените название на `Green Kayak`. Щелкните на кнопке Save; браузер отправит серверу запрос HTTP PUT. Сервер вернет измененный объект, добавленный в массив `products` репозитория, и этот объект будет выведен в таблице (рис. 24.3).

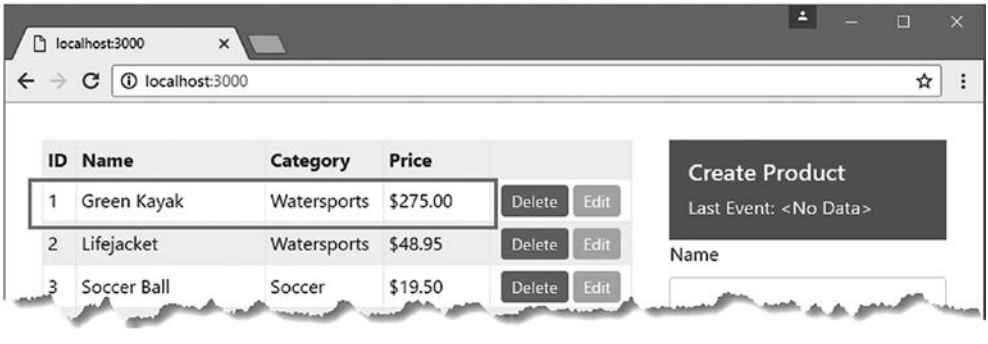


Рис. 24.3. Отправка запроса PUT серверу

Вы можете убедиться в том, что сервер сохранил изменения; введите в браузере адрес *http://localhost/products/1*, для которого будет получено следующее представление объекта:

```
{
  "id": 1,
  "name": "Green Kayak",
  "category": "Watersports",
  "price": 275
}
```

Консолидация запросов HTTP

Все методы класса источника данных строятся по одной базовой схеме: они отправляют запрос HTTP, а затем разбирают ответ сервера в формате JSON. Это означает, что все изменения в механизме отправки запросов HTTP придется по-

вторять в четырех местах для обновления реализации всех команд HTTP — GET, POST, PUT и DELETE.

Класс `Http` определяет метод `request`, который позволяет передать команду HTTP в аргументе. В листинге 24.11 метод `request` используется для консолидации запросов HTTP, чтобы предотвратить дублирование в классе источника данных.

Листинг 24.11. Консолидация запросов HTTP в файле rest.datasource.ts

```
import { Injectable, Inject, OpaqueToken } from "@angular/core";
import { Http, Request, RequestMethod } from "@angular/http";
import { Observable } from "rxjs/Observable";
import { Product } from "./product.model";
import "rxjs/add/operator/map";

export const REST_URL = new OpaqueToken("rest_url");

@Injectable()
export class RestDataSource {

    constructor(private http: Http,
        @Inject(REST_URL) private url: string) { }

    getData(): Observable<Product[]> {
        return this.sendRequest(RequestMethod.Get, this.url);
    }

    saveProduct(product: Product): Observable<Product> {
        return this.sendRequest(RequestMethod.Post, this.url, product);
    }

    updateProduct(product: Product): Observable<Product> {
        return this.sendRequest(RequestMethod.Put,
            `${this.url}/${product.id}`, product);
    }

    deleteProduct(id: number): Observable<Product> {
        return this.sendRequest(RequestMethod.Delete, `${this.url}/${id}`);
    }

    private sendRequest(verb: RequestMethod,
                        url: string, body?: Product): Observable<Product> {
        return this.http.request(new Request({
            method: verb,
            url: url,
            body: body
        })).map(response => response.json());
    }
}
```

Метод `request` получает объект `Request` с описанием отправляемого запроса HTTP. Процесс создания объекта `Request` выглядит немного странно, потому что

он использует интерфейс `RequestArgs` для определения свойств конфигурации, а в аргументах метода `Request` он может выражаться объектным литералом:

```
...
return this.http.request(new Request({
    method: verb,
    url: url,
    body: body
})).map(response => response.json());
...
```

В табл. 24.6 описаны свойства, используемые для настройки запроса.

Таблица 24.6. Свойства ResponseArgs

Имя	Описание
method	Свойство задает команду/метод HTTP, которая будет использоваться для запроса, в виде строки или значения из перечисления RequestMethod
url	Свойство задает URL-адрес для отправки запроса
headers	Свойство возвращает объект Headers для задания заголовков запроса (см. раздел «Настройка заголовков запроса»)
body	Свойство используется для задания тела запроса. Объект, заданный этому свойству, сериализуется в формате JSON при отправке запроса
withCredentials	Если свойство содержит true, при выдаче междоменных запросов включаются cookie аутентификации. Этот параметр должен использоваться только с серверами, включающими в ответы заголовок Access-Control-Allow-Credentials в соответствии со спецификацией CORS (Cross-Origin Resource Sharing). За подробностями обращайтесь к разделу «Создание междоменных запросов»

В листинге свойства `method`, `url` и `body` используются для настройки конфигурации запросов, а использование объекта `Http` консолидируется в одном методе.

Создание междоменных запросов

По умолчанию браузеры устанавливают политику безопасности, которая позволяет коду JavaScript выдавать асинхронные запросы HTTP только с таким же источником, как документ, в котором они содержатся. Эта политика призвана снизить риск *межсайтовых сценарных атак*, которые обманным способом заставляют браузер выполнять вредоносный код. Подробности таких атак выходят за рамки книги, но в статье *http://en.wikipedia.org/wiki/Cross-site_scripting* приведено хорошее введение в тему.

Для разработчиков Angular политика единого домена может создать проблемы при работе с веб-службами, потому что они обычно находятся вне домена, содер-

жащего код JavaScript приложения. Два URL-адреса считаются принадлежащими одному источнику, если они используют один протокол, хост и порт, и считаются принадлежащими к разным источникам в противном случае. URL-адрес файла HTML, содержащего код JavaScript приложения, имеет вид *http://localhost:3000/index.html*. В табл. 24.7 указано, считаются ли принадлежащими к одному или разным источникам URL-адреса, похожие на URL приложения.

Таблица 24.7. URL-адреса и их источники

URL	Сравнение источников
http://localhost:3000/otherfile.html	Тот же источник
http://localhost:3000/app/main.js	Тот же источник
https://localhost:3000/index.html	Разные источники (различаются протоколы)
http://localhost:3500/products	Разные источники (различаются порты)
http://angular.io/index.html	Разные источники (различаются хосты)

Как видно из таблицы, URL-адрес REST-совместимой веб-службы *http://localhost:3500/products* имеет другой источник, потому что в нем используется не тот порт, который используется в основном приложении.

Запросы HTTP, созданные с использованием класса Angular `Http`, автоматически используют механизм CORS (Cross-Origin Resource Sharing) для отправки запросов к другим источникам. В рамках CORS браузер включает в асинхронный запрос HTTP заголовки, которые сообщают серверу источник кода JavaScript. Ответ от сервера включает заголовки, которые сообщают браузеру, принимается ли запрос. Подробности CORS выходят за рамки книги, но введение в тему доступно по адресу *http://www.w3.org/TR/cors*.

Для разработчика Angular поддержка CORS обеспечивается автоматически при условии, что спецификация поддерживается сервером, получающим асинхронные запросы HTTP. Пакет `json-server`, предоставляющий REST-совместимую веб-службу для примеров, поддерживает CORS и принимает запросы от любых источников; именно поэтому наши примеры работают. Если вы хотите увидеть CORS в действии, воспользуйтесь инструментарием разработчика F12 в браузере и понаблюдайте за сетевыми запросами, генерируемыми при редактировании или создании товара. Возможно, вы увидите запрос с командой OPTIONS (так называемый *подготовительный запрос*), который используется браузером для проверки того, что ему разрешено выдавать запросы POST и PUT к серверу. Этот и последующий запрос, отправляющий данные серверу, содержат заголовок Origin, а ответ будет содержать один или несколько заголовков Access-Control-Allow, при помощи которых сервер сообщает, что он готов получить от клиента.

Все это происходит автоматически, а единственным параметром конфигурации является свойство `withCredentials` из табл. 24.6. Если это свойство истинно, то браузер включает cookie аутентификации, а в запрос к серверу будут включены заголовки от источника.

Использование запросов JSONP

Механизм CORS доступен только в том случае, если он поддерживается сервером, которому отправляются запросы HTTP. Для серверов, не поддерживающих CORS, Angular предоставляет поддержку *JSONP* с более ограниченной формой междоменных запросов.

Работа JSONP основана на добавлении в DOM элемента `script`, в атрибуте `src` которого задается междоменный сервер. Браузер отправляет серверу запрос GET, в ответе на который возвращается код JavaScript; при выполнении этого кода приложение получает требуемые данные. Фактически JSONP — это трюк, работающий в обход браузерной политики безопасности единого источника. JSONP может использоваться только для выдачи запросов GET и создает большие риски безопасности, чем CORS. Как следствие, JSONP следует использовать только при недоступности CORS.

Поддержка JSONP в Angular определяется в функциональном модуле `JsonpModule`, который определяется в модуле JavaScript `@angular/http`. Чтобы включить поддержку JSONP, в листинге 24.12 в набор импортирования модуля добавляется `JsonpModule`.

Листинг 24.12. Включение поддержки JSONP в файле model.module.ts

```
import { NgModule } from "@angular/core";
import { HttpModule, JsonpModule } from "@angular/http"
import { Model } from "./repository.model";
import { RestDataSource, REST_URL } from "./rest.datasource";

@NgModule({
    imports: [HttpModule, JsonpModule],
    providers: [Model, RestDataSource,
        { provide: REST_URL, useValue: `http://${location.hostname}:3500/products`
}]
})
export class ModelModule { }
```

Angular предоставляет поддержку JSONP через службу `Jsonp`, определенную в модуле `@angular/http`. Эта служба берет на себя управление HTTP-запросом JSONP и обработку ответа (без этого процесс был бы слишком рутинным и ненадежным). В листинге 24.13 показано, как источник данных использует JSONP для запроса исходных данных приложения.

ПРИМЕЧАНИЕ

Класс Jsonp определяет тот же набор методов, что и класс Http, но поддерживает только запросы GET. При попытке отправить запрос любого другого типа произойдет ошибка.

Листинг 24.13. Создание запроса JSONP в файле rest.datasource.ts

```
import { Injectable, Inject, OpaqueToken } from "@angular/core";
import { Http, Request, RequestMethod, Jsonp } from "@angular/http";
import { Observable } from "rxjs/Observable";
```

```
import { Product } from "./product.model";
import "rxjs/add/operator/map";
export const REST_URL = new OpaqueToken("rest_url");

@Injectable()
export class RestDataSource {

    constructor(private http: Http, private jsonp: Jsonp,
        @Inject(REST_URL) private url: string) { }

    getData(): Observable<Product[]> {
        return this.jsonp.get(this.url + "?callback=JSONP_CALLBACK")
            .map(response => response.json());
    }

    saveProduct(product: Product): Observable<Product> {
        return this.sendRequest(RequestMethod.Post, this.url, product);
    }

    updateProduct(product: Product): Observable<Product> {
        return this.sendRequest(RequestMethod.Put,
            `${this.url}/${product.id}`, product);
    }

    deleteProduct(id: number): Observable<Product> {
        return this.sendRequest(RequestMethod.Delete, `${this.url}/${id}`);
    }

    private sendRequest(verb: RequestMethod,
        url: string, body?: Product): Observable<Product> {
        return this.http.request(new Request({
            method: verb,
            url: url,
            body: body
        })).map(response => response.json());
    }
}
```

При использовании JSONP запрашиваемый URL-адрес должен включать параметр `callback`, которому присваивается значение `JSONP_CALLBACK`:

```
...
return this.jsonp.get(this.url + "?callback=JSONP_CALLBACK")
    .map(response => response.json());
...
```

Создавая запрос HTTP, Angular заменяет `JSONP_CALLBACK` именем динамически сгенерированной функции. Присмотревшись к сетевым запросам, создаваемым браузером, вы увидите, что исходный запрос отправляется на URL следующего вида:

```
http://localhost:3500/products?callback=__ng_jsonp__.__req0.finished
```

Сервер использует значение параметра `callback` для генерирования файла JavaScript, который вызывает указанную функцию и передает ей данные от модели:

```
typeof __ng_jsonp__.__req0.finished === 'function'
     &&__ng_jsonp__.__req0.finished([
  { "id": 1, "name": "Green Kayak", "category": "Watersports", "price": 275 },
  { "id": 2,"name": "Lifejacket", "category": "Watersports", "price": 48.95 },
  { ...другие объекты опущены для краткости... }
]);
```

Браузер загружает файл JavaScript, сгенерированный сервером, и выполняет функцию, которая используется Angular для получения данных в приложении. JSONP — более ограниченный механизм создания междоменных запросов, и, в отличие от CORS, он обходит политику безопасности браузера, но в крайнем случае он может оказаться полезным запасным вариантом.

Настройка заголовков запроса

Если вы используете коммерческую REST-совместимую веб-службу, вам часто придется задавать заголовок запроса для передачи ключа API, чтобы сервер мог связать запрос с вашим приложением для управления доступом и выставления счетов. Чтобы задать этот (или любой другой) заголовок, вы можете настроить объект конфигурации `RequestArgs`, который передается методу `request` (листинг 24.14). Также в этом листинге мы возвращаемся к использованию для всех запросов класса `Http` (вместо JSONP).

Листинг 24.14. Назначение заголовка запроса в файле rest.datasource.ts

```
import { Injectable, Inject, OpaqueToken } from "@angular/core";
import { Http, Request, RequestMethod, Headers } from "@angular/http";
import { Observable } from "rxjs/Observable";
import { Product } from "./product.model";
import "rxjs/add/operator/map";

export const REST_URL = new OpaqueToken("rest_url");

@Injectable()
export class RestDataSource {

    constructor(private http: Http,
        @Inject(REST_URL) private url: string) { }

    getData(): Observable<Product[]> {
        return this.sendRequest(RequestMethod.Get, this.url);
    }

    saveProduct(product: Product): Observable<Product> {
        return this.sendRequest(RequestMethod.Post, this.url, product);
    }
```

```
    updateProduct(product: Product): Observable<Product> {
        return this.sendRequest(RequestMethod.Put,
            `${this.url}/${product.id}`, product);
    }

    deleteProduct(id: number): Observable<Product> {
        return this.sendRequest(RequestMethod.Delete, `${this.url}/${id}`);
    }
    private sendRequest(verb: RequestMethod,
        url: string, body?: Product): Observable<Product> {
        return this.http.request(new Request({
            method: verb,
            url: url,
            body: body,
            headers: new Headers({
                "Access-Key": "<secret>",
                "Application-Name": "exampleApp"
            })
        })).map(response => response.json());
    }
}
```

Свойству `headers` назначается объект `Headers`, который может быть создан на базе объекта `map` со свойствами, соответствующими именам заголовков, и значениями, которые должны быть им присвоены. Воспользовавшись инструментарием разработчика F12 для анализа асинхронных запросов HTTP, вы увидите, что среди стандартных заголовков, создаваемых браузером, встречаются два заголовка, указанные в листинге:

```
...
Accept:*/*
Accept-Encoding:gzip, deflate, sdch, br
Accept-Language:en-US,en;q=0.8
access-key:<secret>
application-name:exampleApp
Connection:keep-alive
...
```

Имена заголовков передаются в нижнем регистре, но в именах заголовков HTTP регистр не учитывается. Если вам потребуются более сложные требования к заголовкам запросов, используйте методы, определенные классом `Headers` и перечисленные в табл. 24.8.

Таблица 24.8. Методы Headers

Имя	Описание
get(name)	Возвращает первое значение указанного заголовка
getAll(name)	Возвращает все значения указанного заголовка

Имя	Описание
has(name)	Возвращает true, если коллекция содержит указанный заголовок
set(header, value)	Заменяет все существующие значения указанного заголовка одним значением
set(header, values)	Заменяет все существующие значения указанного заголовка массивом значений
append(name, value)	Присоединяет значение к списку значений указанного заголовка
delete(name)	Удаляет указанный заголовок из коллекции
toJson()	Возвращает представление всех заголовков и значений в формате JSON

Заголовки HTTP могут содержать множественные значения, этим объясняется наличие методов, которые присоединяют значения к заголовкам или заменяют все значения в коллекции. Листинг 24.15 создает пустой объект `Headers` и заполняет его заголовками с множественными значениями.

Листинг 24.15. Назначение множественных значений заголовков в файле rest.datasource.ts

```
...
private sendRequest(verb: RequestMethod,
        url: string, body?: Product): Observable<Product> {

    let headers = new Headers();
    headers.set("Access-Key", "<secret>");
    headers.set("Application-Names", ["exampleApp", "proAngular"]);

    return this.http.request(new Request({
        method: verb,
        url: url,
        body: body,
        headers: headers
    })).map(response => response.json());
}
...
```

Отправляя запросы серверу, браузер включает в него следующие заголовки:

```
...
Accept:*/*
Accept-Encoding:gzip, deflate, sdch, br
Accept-Language:en-US,en;q=0.8
access-key:<secret>
application-names:exampleApp,proAngular
Connection:keep-alive
...
```

Обработка ошибок

На данный момент в приложении полностью отсутствует обработка ошибок; это означает, что Angular не будет знать, как поступать при возникновении проблем с запросами HTTP. Чтобы в приложении было проще сгенерировать ошибку, добавьте в таблицу товаров кнопку для создания запроса HTTP на удаление объекта, не существующего на сервере (листинг 24.16).

Листинг 24.16. Добавление кнопки для генерирования ошибки в файл table.component.html

```
<table class="table table-sm table-bordered table-striped">
    <tr>
        <th>ID</th><th>Name</th><th>Category</th><th>Price</th><th></th>
    </tr>
    <tr *ngFor="let item of getProducts()">
        <td style="vertical-align:middle">{{item.id}}</td>
        <td style="vertical-align:middle">{{item.name}}</td>
        <td style="vertical-align:middle">{{item.category}}</td>
        <td style="vertical-align:middle">
            {{item.price | currency:"USD":true }}
        </td>
        <td class="text-xs-center">
            <button class="btn btn-danger btn-sm" (click)="deleteProduct(item.id)">
                Delete
            </button>
            <button class="btn btn-warning btn-sm" (click)="editProduct(item.id)">
                Edit
            </button>
        </td>
    </tr>

</table>
<button class="btn btn-primary" (click)="createProduct()">
    Create New Product
</button>
<button class="btn btn-danger" (click)="deleteProduct(-1)">
    Generate HTTP Error
</button>
```

Элемент `button` вызывает метод `deleteProduct` компонента с аргументом -1. Компонент приказывает репозиторию удалить этот объект, что приведет к отправке запроса HTTP DELETE для несуществующего пути `/products/-1`. Если вы откроете консоль JavaScript и щелкнете на новой кнопке, открывается ответ от сервера:

```
EXCEPTION: Response with status: 404 Not Found for URL: http://localhost:3500/
products/-1
```

Как улучшить ситуацию? Вы должны обнаруживать такие ошибки и оповещать о них пользователя, который обычно не смотрит на консоль JavaScript. Реальное приложение также может реагировать на ошибки, сохраняя информацию о них

в журнале для последующего анализа, но мы будем действовать проще и выводить сообщение об ошибке.

Генерирование сообщений для пользователя

Обработка ошибки должна начинаться с преобразования исключения HTTP в сообщение, которое будет понятно пользователю. Сообщение об ошибке по умолчанию — то, что выводится на консоль JavaScript, — содержит слишком много информации для пользователя. Пользователю не нужно знать URL-адрес, на который был отправлен запрос; достаточно, если он будет примерно понимать суть возникшей проблемы.

Для преобразования сообщений лучше всего воспользоваться методами `catch` и `throw`, определяемыми классом `Observable`. Метод `catch` используется для перехвата ошибок, происходящих в последовательности `Observable`, а метод `throw` создает новый объект `Observable`, который просто содержит ошибку. В листинге 24.17 оба метода применяются к источнику данных.

ПРИМЕЧАНИЕ

Обратите внимание: метод throw импортируется из rxjs/add/observable.

Листинг 24.17. Преобразование ошибок в файле rest.datasource.ts

```
import { Injectable, Inject, OpaqueToken } from "@angular/core";
import { Http, Request, RequestMethod, Headers, Response } from "@angular/http";
import { Observable } from "rxjs/Observable";
import { Product } from "./product.model";
import "rxjs/add/operator/map";
import "rxjs/add/operator/catch";
import "rxjs/add/observable/throw";

export const REST_URL = new OpaqueToken("rest_url");

@Injectable()
export class RestDataSource {

    constructor(private http: Http,
        @Inject(REST_URL) private url: string) { }

    getData(): Observable<Product[]> {
        return this.sendRequest(RequestMethod.Get, this.url);
    }

    saveProduct(product: Product): Observable<Product> {
        return this.sendRequest(RequestMethod.Post, this.url, product);
    }

    updateProduct(product: Product): Observable<Product> {
        return this.sendRequest(RequestMethod.Put,
            `${this.url}/${product.id}`, product);
```

```
    }

    deleteProduct(id: number): Observable<Product> {
        return this.sendRequest(RequestMethod.Delete, `${this.url}/${id}`);
    }

    private sendRequest(verb: RequestMethod,
        url: string, body?: Product): Observable<Product> {
        let headers = new Headers();
        headers.set("Access-Key", "<secret>");
        headers.set("Application-Names", ["exampleApp", "proAngular"]);

        return this.http.request(new Request({
            method: verb,
            url: url,
            body: body,
            headers: headers
        })).map(response => response.json())
            .catch((error: Response) => Observable.throw(
                `Network Error: ${error.statusText} (${error.status})`));
    }
}
```

Функция, передаваемая методу `catch`, вызывается при возникновении ошибки и получает объект `Response` с описанием результата. Метод `Observable.throw` создает новый объект `Observable`, содержащий только объект ошибки, который в данном случае используется для генерирования сообщения, содержащего код статуса HTTP и текст из ответа.

Если сохранить изменения, а потом снова щелкнуть на кнопке Generate HTTP Error, сообщение об ошибке по-прежнему будет выведено на консоль JavaScript в браузере, но в измененном формате, созданном методами `catch/throw`:

```
EXCEPTION: Network Error: Not Found (404)
```

Обработка ошибок

Ошибки были преобразованы, но не обработаны; именно поэтому они продолжают выводиться на консоль JavaScript в браузере в виде исключений.

Существует два способа обработки ошибок. Первый — предоставить функцию обработки ошибок методу `subscribe` для объектов `Observable`, созданных объектом `Http`. Он позволяет локализовать ошибку и дать репозиторию возможность повторить операцию или попытаться восстановить работоспособность иным способом.

Второй вариант основан на замене встроенного механизма обработки ошибок Angular, который реагирует на любые необработанные ошибки в приложении и по умолчанию записывает их на консоль. Именно этот механизм выводит сообщения об ошибках в предыдущих разделах.

Для нашего примера обработчик ошибок по умолчанию будет заменен другим, использующим службу сообщений. Создайте файл errorHandler.ts в папке exampleApp/app/messages и включите в него определение класса из листинга 24.18.

Листинг 24.18. Содержимое файла errorHandler.ts в папке exampleApp/app/messages

```
import { ErrorHandler, Injectable } from "@angular/core";
import { MessageService } from "./message.service";
import { Message } from "./message.model";

@Injectable()
export class MessageErrorHandler implements ErrorHandler {

    constructor(private messageService: MessageService) {
    }

    handleError(error) {
        let msg = error instanceof Error ? error.message : error.toString();
        setTimeout(() => this.messageService
            .reportMessage(new Message(msg, true)), 0);
    }
}
```

Класс `ErrorHandler` определяется в модуле `@angular/core` и реагирует на ошибки через метод `handleError`. Класс, показанный в листинге, заменяет реализацию этого метода по умолчанию другой, использующей `MessageService` для вывода сообщения об ошибке. Функция `setTimeout` гарантирует, что сообщение будет выведено для пользователя; она помогает обойти странность обработки обновлений в Angular.

Чтобы заменить `ErrorHandler` по умолчанию, мы воспользуемся провайдером класса в модуле сообщений (листинг 24.19).

Листинг 24.19. Замена обработчика ошибок по умолчанию в файле message.module.ts

```
import { NgModule, ErrorHandler } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { MessageComponent } from "./message.component";
import { MessageService } from "./message.service";
import { MessageErrorHandler } from "./errorHandler";

@NgModule({
    imports: [BrowserModule],
    declarations: [MessageComponent],
    exports: [MessageComponent],
    providers: [MessageService,
        { provide: ErrorHandler, useClass: MessageErrorHandler }]
})
export class MessageModule { }
```

Функция обработки ошибок использует `MessageService` для выдачи сообщения об ошибке для пользователя. После того как изменения будут сохранены, кнопка Generate HTTP Error создает ошибку, которую увидит пользователь (рис. 24.4).

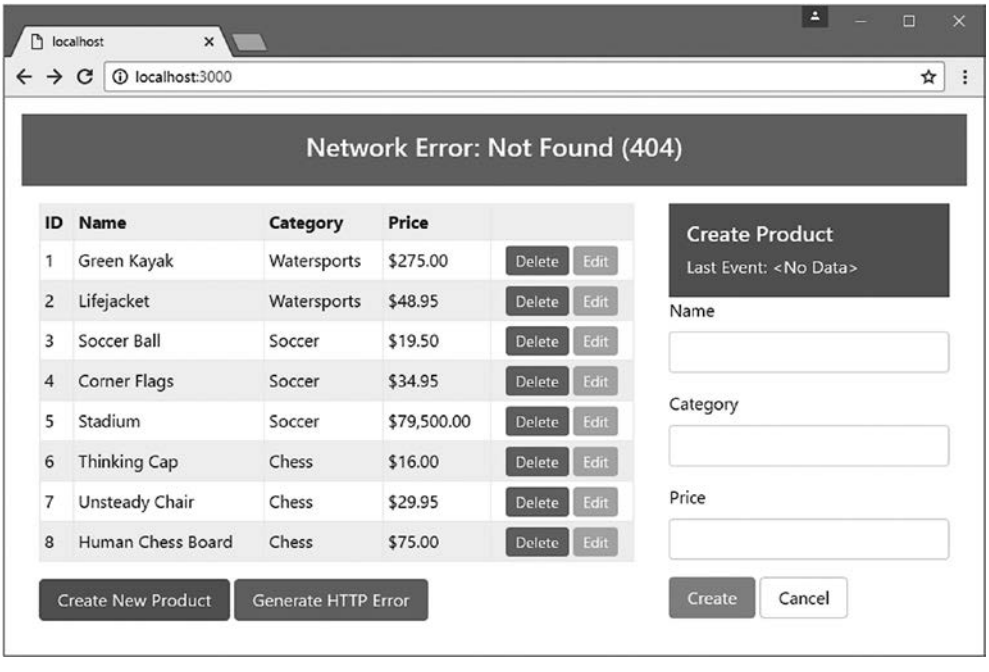


Рис. 24.4. Обработка ошибки HTTP

Итоги

В этой главе вы узнали, как создавать асинхронные запросы HTTP в приложениях Angular. Я описал REST-совместимые веб-службы и методы класса Angular `Http`, которые могут использоваться для взаимодействия с ними. Я объяснил, как браузер ограничивает запросы к разным источникам и как Angular поддерживает механизмы CORS и JSONP для выдачи запросов за пределы источника приложения. В следующей главе будет представлен механизм маршрутизации URL, обеспечивающий навигацию в сложных приложениях.

25 Маршрутизация и навигация: часть 1

Механизм *маршрутизации* (routing) в Angular позволяет приложениям менять компоненты и шаблоны, отображаемые для пользователя, в соответствии с изменениями URL в браузере. У разработчика появляется возможность создавать сложные приложения с открытым и гибким механизмом адаптации к отображаемому контенту при минимальном объеме кода. Для ее поддержки существуют привязки данных и службы, которые могут изменять URL в браузере для навигации пользователя в приложении.

Маршрутизация особенно полезна с ростом сложности проекта, потому что она позволяет определять структуру приложения отдельно от компонентов и директив; это означает, что структурные изменения могут вноситься в конфигурации маршрутизации без применения их к отдельным компонентам.

В этой главе я покажу, как работает базовая система маршрутизации, и применю ее в нашем приложении. В главах 26 и 27 рассматриваются расширенные возможности маршрутизации. В табл. 25.1 маршрутизация представлена в контексте.

Таблица 25.1. Маршрутизация в контексте

Вопрос	Ответ
Что это такое?	Механизм маршрутизации использует URL-адрес в браузере для управления отображаемым контентом
Для чего они нужны?	Маршрутизация позволяет отделить структуру приложения от компонентов и шаблонов приложения. Изменения в структуре приложения вносятся в конфигурации маршрутизации, а не в отдельных компонентах и директивах
Как они используются?	Конфигурация маршрутизации определяется в виде набора фрагментов, которые используются для поиска соответствий в URL браузера и выбора компонентов, шаблоны которых должны отображаться как контент элемента HTML с именем router-outlet
Есть ли у них недостатки или скрытые проблемы?	Конфигурация маршрутизации может выйти из-под контроля, особенно если схема URL определяется по мере надобности без предварительного планирования
Есть ли альтернативы?	Использовать маршрутизацию не обязательно. Чтобы добиться аналогичного результата, можно создать компонент, представление которого выбирает отображаемый контент при помощи директивы ngFor или ngSwitch. С другой стороны, такие решения становятся более сложными, чем решения на базе маршрутизации, с ростом размера и сложности приложения

В табл. 25.2 приведена краткая сводка материала главы.

ПРИМЕЧАНИЕ

Для работы примеров этой главы необходим модуль Reactive Extensions, созданный с использованием пакета systemjs-builder (см. главу 22). Если вы еще не создали этот файл, сделайте это сейчас или загрузите проект этой главы с сайта apress.com.

Таблица 25.2. Сводка материала главы

Проблема	Решение	Листинг
Использование URL для выбора отображаемого контента	Используйте маршрутизацию URL	1–9
Навигация с использованием элемента HTML	Примените атрибут routerLink	10–12
Реакция на изменение маршрута	Используйте службы маршрутизации для получения уведомлений	13
Включение информации в URL	Используйте параметры маршрута	14–20
Навигация в программном коде	Используйте службу Router	21
Получение уведомлений о маршрутизации	Обрабатывайте события маршрутизации	22–23

Подготовка проекта

Функциональность, рассмотренная в этой главе, зависит от модуля Angular `Router`; этот модуль необходимо добавить в приложение. В листинге 25.1 продемонстрировано включение модуля в список пакетов.

ВНИМАНИЕ

Обратите внимание: номер версии этого пакета выше, чем у других пакетов Angular. Эта странность возникает из-за того, что механизм маршрутизации был существенно переработан во время разработки Angular 2.

Листинг 25.1. Добавление пакета Router в файл package.json

```
{
  "dependencies": {
    "@angular/common": "2.2.0",
    "@angular/compiler": "2.2.0",
    "@angular/core": "2.2.0",
    "@angular/platform-browser": "2.2.0",
    "@angular/platform-browser-dynamic": "2.2.0",
    "@angular/upgrade": "2.2.0",
    "@angular/forms": "2.2.0",
    "@angular/http": "2.2.0",
    "@angular/router": "3.2.0",
```

```
    "reflect-metadata": "0.1.8",
    "rxjs": "5.0.0-beta.12",
    "zone.js": "0.6.26",
    "core-js": "2.4.1",
    "classlist.js": "1.1.20150312",
    "systemjs": "0.19.40",
    "bootstrap": "4.0.0-alpha.4",
    "intl": "1.2.4",
    "html5-history-api": "4.2.7"
  },

  "devDependencies": {
    "lite-server": "2.2.2",
    "typescript": "2.0.2",
    "typings": "1.3.2",
    "concurrently": "2.2.0",
    "systemjs-builder": "0.15.32",
    "json-server": "0.8.21"
  },

  "scripts": {
    "start": "concurrently \"npm run tscwatch\" \"npm run lite\" \"npm run json\"
",
    "tsc": "tsc",
    "tscwatch": "tsc -w",
    "lite": "lite-server",
    "json": "json-server --p 3500 restData.js",
    "typings": "typings",
    "postinstall": "typings install"
  }
}
```

Механизм маршрутизации Angular зависит от JavaScript History API, который не поддерживается старыми браузерами, поэтому мы добавим пакет `html5-history-api` в приложение, чтобы предоставить полизаполнение для отсутствующего API.

ПРИМЕЧАНИЕ

Проекты всех этих глав можно загрузить с сайта apress.com.

Модуль маршрутизации Angular также необходимо зарегистрировать в загрузчике модулей JavaScript SystemJS (листинг 25.2).

Листинг 25.2. Добавление модуля Router в файл systemjs.config.js

```
(function (global) {

    var paths = {
        "@angular/*": "node_modules/@angular/*"
    }

    var packages = { "app": {} };

    var angularModules = ["common", "compiler",
```

```
     "core", "platform-browser", "platform-browser-dynamic", "forms",
     "http", "router"];

    angularModules.forEach(function (pkg) {
        packages["@angular/" + pkg] = {
            main: "/bundles/" + pkg + ".umd.min.js"
        };
    });

    System.config({ paths: paths, packages: packages });

})(this);
```

Блокировка вывода событий изменения состояния

Приложение настроено для вывода событий изменения состояния, отправляемых компонентом таблицы, в двух местах: через службу сообщений и в шаблоне компонента формы. Теперь эти сообщения не нужны, и в листинге 25.3 вывод событий исключается из шаблона компонента.

Листинг 25.3. Блокировка вывода событий в файле form.component.html

```
<div class="bg-primary p-a-1" [class.bg-warning]="editing">
    <h5>{{editing  ? "Edit" : "Create"}} Product</h5>
    <!--Last Event: {{ stateEvents | async | formatState }}-->
</div>

<form novalidate #form="ngForm" (ngSubmit)="submitForm(form)" (reset)="resetForm()"
>

    <div class="form-group">
        <label>Name</label>
        <input class="form-control" name="name"
               [(ngModel)]="product.name" required />
    </div>

    <div class="form-group">
        <label>Category</label>
        <input class="form-control" name="category"
               [(ngModel)]="product.category" required />
    </div>

    <div class="form-group">
        <label>Price</label>
        <input class="form-control" name="price"
               [(ngModel)]="product.price"
               required pattern="^[0-9\.]+$" />
    </div>

    <button type="submit" class="btn btn-primary"
            [class.btn-warning]="editing" [disabled]="form.invalid">
        {{editing ? "Save" : "Create"}}
    </button>
    <button type="reset" class="btn btn-secondary">Cancel</button>
</form>
```

В листинге 25.4 исключается код, передающий события изменения состояния службе сообщений.

Листинг 25.4. Блокировка событий изменения состояния в файле core.model.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule } from "@angular/forms";
import { ModelModule, Model } from "model/module";
import { TableComponent } from "./table.component";
import { FormComponent } from "./form.component";
import { Product } from "model/module";
import { SharedState, SHARED_STATE } from "./sharedState.model";
import { Subject } from "rxjs/Subject";
import { StatePipe } from "./state.pipe";
import { MessageModule, MessageService, Message } from "messages/module";
import { MODES } from "./sharedState.model";

@NgModule({
    imports: [BrowserModule, FormsModule, ModelModule, MessageModule],
    declarations: [TableComponent, FormComponent, StatePipe],
    exports: [ModelModule, TableComponent, FormComponent],
    providers: [{
        provide: SHARED_STATE,
        deps: [MessageService, Model],
        useFactory: (messageService, model) => {
            return new Subject<SharedState>();
            //subject.subscribe(m => messageService.reportMessage(
            //    new Message(MODES[m.mode] + (m.id != undefined
            //        ? ` ${model.getProduct(m.id).name}` : "")))
            //);
            //return subject;
        }
    }]
})
export class CoreModule { }
```

Сохраните изменения и выполните следующую команду из папки exampleApp, чтобы загрузить и установить модуль Angular:

```
npm install
```

После того как обновление будет завершено, выполните следующую команду из папки exampleApp, чтобы запустить компилятор TypeScript, сервер HTTP для разработки и REST-совместимую веб-службу:

```
npm start
```

Открывается новое окно (или вкладка) браузера с контентом, показанным на рис. 25.1.

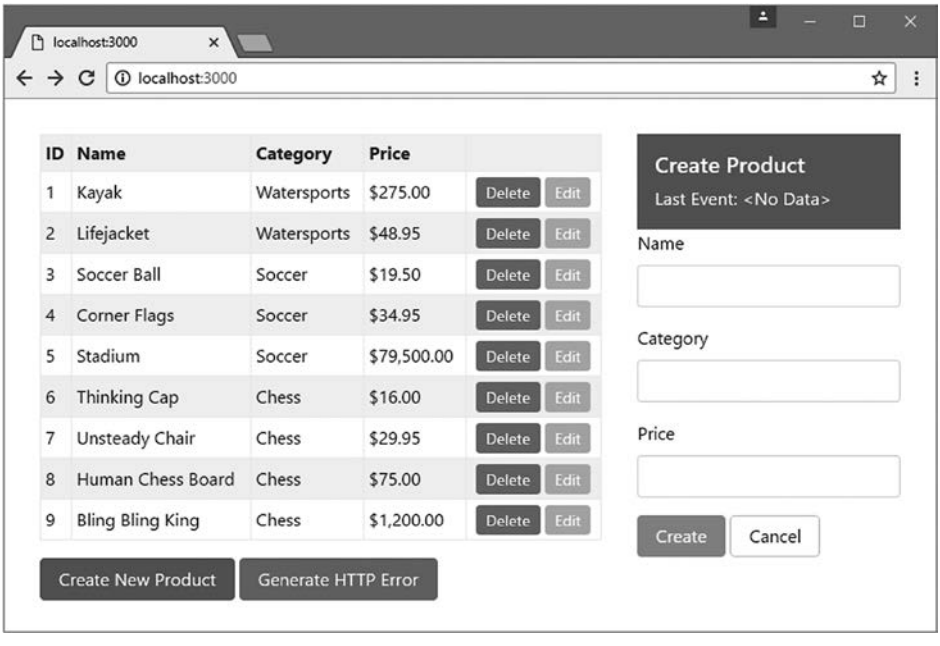


Рис. 25.1. Запуск приложения

Знакомство с маршрутизацией

На данный момент весь контент приложения всегда остается видимым для пользователя. В нашем примере это означает, что и таблица, и форма постоянно видны, и пользователь должен сам следить за тем, какая часть приложения используется для текущей задачи.

Для простых приложений это нормально, но в сложном проекте с множеством функциональных областей ситуация может выйти из-под контроля, если все они отображаются одновременно. Маршрутизация URL структурирует приложение при помощи естественного, хорошо понятного аспекта веб-приложений — URL-адреса. В этом разделе мы добавим маршрутизацию URL в приложение, чтобы видимой была либо таблица, либо форма, а активный компонент выбирался в зависимости от действий пользователя. Этот пример позволит объяснить принципы работы маршрутизации и заложит основу для более сложной функциональности.

Создание конфигурации маршрутизации

Применение маршрутизации начинается с определения *маршрутов* (routes) — соответствий между URL и компонентами, которые должны отображаться для пользователя. Конфигурации маршрутизации обычно определяются в файле с именем app.routing.ts, находящемся в папке app. Создайте этот файл и добавьте в него команды из листинга 25.5.

Листинг 25.5. Содержимое файла app.routing.ts в папке exampleApp/app

```
import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "./core/table.component";
import { FormComponent } from "./core/form.component";

const routes: Routes = [
    { path: "form/edit", component: FormComponent },
    { path: "form/create", component: FormComponent },
    { path: "", component: TableComponent }]

export const routing = RouterModule.forRoot(routes);
```

Класс `Routes` определяет коллекцию маршрутов, каждый из которых сообщает Angular, как обрабатывать конкретный URL-адрес. В этом примере используются простейшие свойства: свойство `path` задает URL, а свойство `component` — компонент, который будет отображаться для пользователя.

Свойство `path` задается относительно остального кода приложения; это означает, что конфигурация из листинга 25.5 создаст маршруты, представленные в табл. 25.3.

Таблица 25.3. Маршруты, создаваемые в примере

URL	Отображаемый компонент
http://localhost:3000/form/edit	FormComponent
http://localhost:3000/form/create	FormComponent
http://localhost:3000/	TableComponent

Маршруты упакованы в модуль при помощи метода `RouterModule.forRoot`. Метод `forRoot` создает модуль, включающий службу маршрутизации. Также существует метод `forChild`, который не включает службу (см. главу 26, где я объясняю, как создавать маршруты для функциональных модулей).

И хотя свойства `path` и `component` чаще всего используются при определении маршрутов, существует ряд дополнительных свойств, которые могут использоваться для определения маршрутов с расширенными возможностями. Эти свойства описаны в табл. 25.4 с указанием глав, в которых они рассматриваются более подробно.

Таблица 25.4. Свойства Routes, используемые для определения маршрутов

Имя	Описание
path	Свойство задает путь для маршрута
component	Свойство задает компонент, который должен выбираться в том случае, если активизируемый URL соответствует path
pathMatch	Свойство сообщает Angular, как следует проверять соответствие текущего URL свойству path. Допустимы два значения: full (значение path должно полностью соответствовать URL) и prefix (значение path считается соответствующим, даже если URL включает дополнительные сегменты, не входящие в значение path). Свойство является обязательным при использовании свойства redirectTo (см. главу 26)

Таблица 25.4 (*окончание*)

Имя	Описание
redirectTo	Свойство используется для создания маршрута, перенаправляющего браузер на другой URL. За подробностями обращайтесь к главе 26
children	Свойство используется для задания дочерних маршрутов, которые отображают дополнительные компоненты во вложенных элементах router-outlet, содержащихся в шаблоне компонента активизации (см. главу 26)
outlet	Свойство используется для поддержки множественных элементов outlet (см. главу 27)
resolve	Свойство используется для определения работы, которая должна быть завершена перед активизацией маршрута (см. главу 27)
canActivate	Свойство используется для управления тем, когда может активизироваться маршрут (см. главу 27)
canActivateChild	Свойство используется для управления тем, когда может активизироваться дочерний маршрут (см. главу 27)
canDeactivate	Свойство используется для управления тем, когда маршрут может деактивизироваться для активизации нового маршрута (см. главу 27)
loadChildren	Свойство используется для настройки модуля, который загружается только в случае необходимости (см. главу 27)
canLoad	Свойство используется для управления загрузкой модулей по требованию

ПОРЯДОК ОПРЕДЕЛЕНИЯ МАРШРУТОВ

Порядок определения маршрутов играет важную роль. Angular сравнивает URL-адрес, по которому перешел браузер, со свойством path каждого маршрута, пока не найдет соответствие. Это означает, что самые конкретные маршруты должны определяться в первую очередь, а более универсальные маршруты должны располагаться ближе к концу списка. Для маршрутов из листинга 25.5 это несущественно, но порядок определения становится важным при использовании параметров маршрута (см. раздел «Использование параметров маршрутов» этой главы) или добавлении дочерних маршрутов (см. главу 26).

Если окажется, что ваша конфигурация маршрутизации не приводит к желаемому результату, начните с проверки порядка определения маршрутов.

Создание компонента маршрутизации

При использовании маршрутизации корневой компонент обеспечивает навигацию между разными частями приложения. Создайте файл app.component.ts в папке exampleApp/app и включите в него определение компонента из листинга 25.6.

Листинг 25.6. Содержимое файла app.component.ts в папке exampleApp/app

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

@Component({
    selector: "app",
    moduleId: module.id,
    templateUrl: "app.component.html"
})
export class AppComponent { }
```

Чтобы создать шаблон, связанный с компонентом, создайте файл app.component.html в папке exampleApp/app и включите в него элементы из листинга 25.7.

Листинг 25.7. Содержимое файла app.component.html в папке exampleApp/app

```
<paMessages></paMessages>
<router-outlet></router-outlet>
```

Элемент `paMessages` выводит все информационные сообщения и ошибки приложения. Для целей маршрутизации важен элемент `router-outlet`, потому что он сообщает Angular, где должен отображаться компонент, соответствующий конфигурации маршрутизации.

Обновление корневого модуля

Следующий шагом должно стать обновление корневого модуля, чтобы для начальной загрузки приложения использовался новый корневой компонент (листинг 25.8), который также импортирует модуль, содержащий конфигурацию маршрутизации.

Листинг 25.8. Включение маршрутизации в файле app.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { ModelModule } from "./model/model.module";
import { CoreModule } from "./core/core.module";
import { TableComponent } from "./core/table.component";
import { FormComponent } from "./core/form.component";
import { MessageModule } from "./messages/message.module";
import { MessageComponent } from "./messages/message.component";
import { routing } from "./app.routing";
import { AppComponent } from "./app.component";

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

Завершение конфигурации

Для применения маршрутизации в приложении остается лишь обновить файл index.html (листинг 25.9).

Листинг 25.9. Настройка маршрутизации в файле index.html

```
<!DOCTYPE html>
<html>
<head>
    <base href="/">
    <title></title>
    <meta charset="utf-8" />
    <script src="node_modules/html5-history-api/history.min.js"></script>
    <script src="node_modules/classlist.js/classList.min.js"></script>
    <script src="node_modules/core-js/client/shim.min.js"></script>
    <script src="node_modules/intl/dist/Intl.complete.js"></script>
    <script src="node_modules/zone.js/dist/zone.min.js"></script>
    <script src="node_modules/reflect-metadata/Reflect.js"></script>
    <script src="node_modules/systemjs/dist/system.src.js"></script>
    <script src="rxjs.module.min.js"></script>
    <script src="systemjs.config.js"></script>
    <script>
      System.import("app/main").catch(function(err){ console.error(err); });
    </script>
    <link href="node_modules/bootstrap/dist/css/bootstrap.min.css"
        rel="stylesheet" />
</head>
<body class="m-a-1">
    <app></app>
</body>
</html>
```

Элемент `base` задает URL, с которым будут сравниваться пути маршрутизации из листинга 25.5. Элемент `app` применяет новый корневой компонент, шаблон которого содержит элемент `router-outlet`. Новый элемент `script` в листинге загружает полизаполнение для History API, чтобы приложение правильно работало в старых браузерах.

Когда вы сохраните изменения, а браузер перезагрузит приложение, вы увидите только таблицу товаров (рис. 25.2). URL приложения по умолчанию *http://localhost:3000* соответствует маршруту, который отображает таблицу товаров.

Добавление навигационных ссылок

Базовая конфигурация маршрутизации готова, но механизма навигации в приложении нет: при нажатии кнопки Create New Product или Edit ничего не происходит.

На следующем шаге в приложение добавляются ссылки, которые изменяют URL браузера и при этом инициируют изменения в маршрутизации для вывода другого компонента для пользователя. В листинге 25.10 эти ссылки добавляются в шаблон компонента таблицы.

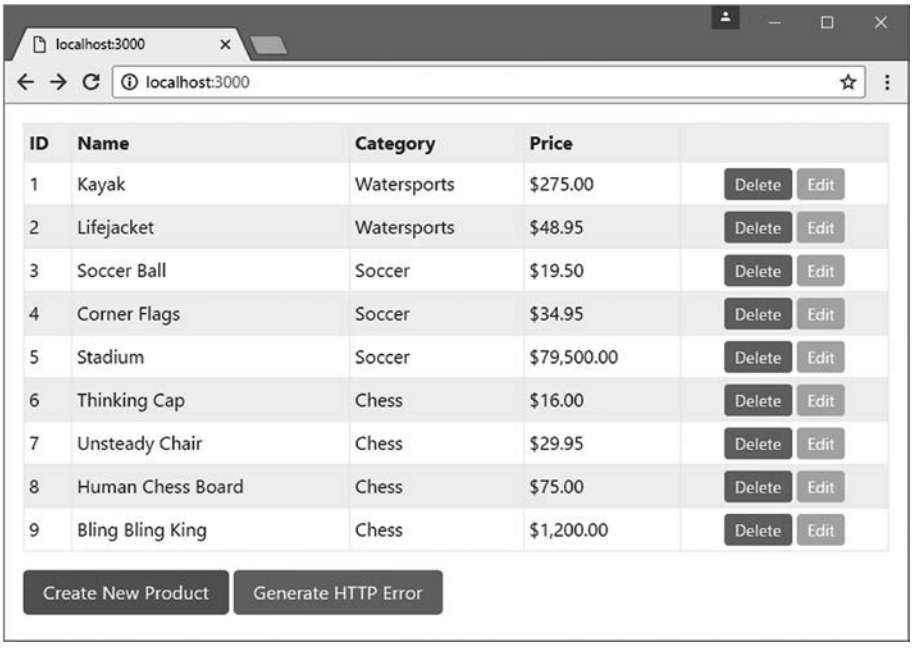


Рис. 25.2. Применение маршрутизации для вывода компонентов

Листинг 25.10. Добавление навигационных ссылок в файл table.component.html

```
<table class="table table-sm table-bordered table-striped">
    <tr>
        <th>ID</th><th>Name</th><th>Category</th><th>Price</th><th></th>
    </tr>
    <tr *ngFor="let item of getProducts()">
        <td style="vertical-align:middle">{{item.id}}</td>
        <td style="vertical-align:middle">{{item.name}}</td>
        <td style="vertical-align:middle">{{item.category}}</td>
        <td style="vertical-align:middle">
            {{item.price | currency:"USD":true }}
        </td>
        <td class="text-xs-center">
            <button class="btn btn-danger btn-sm" (click)="deleteProduct(item.id)">
                Delete
            </button>
            <button class="btn btn-warning btn-sm" (click)="editProduct(item.id)"
                    routerLink="/form/edit">
                Edit
            </button>
        </td>
    </tr>
</table>
<button class="btn btn-primary" (click)="createProduct()" routerLink="/form/
                                        create">
    Create New Product
```

```
</button>
<button class="btn btn-danger" (click)="deleteProduct(-1)">
    Generate HTTP Error
</button>
```

Атрибут `routerLink` применяет директиву из пакета маршрутизации, которая реализует изменение в навигации. Эта директива может применяться к любому элементу, хотя обычно она применяется к элементам `button` и `anchor` (`a`). Выражение директивы `routeLink`, примененной к кнопкам Edit, приказывает Angular выбрать в качестве целевого маршрут `/form/edit`:

```
...
<button class="btn btn-warning btn-sm" (click)="editProduct(item.id)"
        routerLink="/form/edit">
    Edit
</button>
...
```

Та же директива, примененная к кнопке Create New Product, приказывает Angular выбрать в качестве целевого маршрут `/create`.

```
...
<button class="btn btn-primary" (click)="createProduct()" routerLink="/form/
create">
    Create New Product
</button>
...
```

Ссылки маршрутизации, добавленные в шаблон компонента таблицы, позволят пользователю перейти к форме. Изменения в компоненте формы, показанные в листинге 25.11, позволяют пользователю вернуться обратно кнопкой Cancel.

Листинг 25.11. Добавление навигационной ссылки в файл form.component.html

```
<div class="bg-primary p-a-1" [class.bg-warning]="editing">
    <h5>{{editing  ? "Edit" : "Create"}} Product</h5>
</div>
<form novalidate #form="ngForm" (ngSubmit)="submitForm(form)" (reset)="resetForm()" >

    <div class="form-group">
        <label>Name</label>
        <input class="form-control" name="name"
               [(ngModel)]="product.name" required />
    </div>

    <div class="form-group">
        <label>Category</label>
        <input class="form-control" name="category"
               [(ngModel)]="product.category" required />
    </div>

    <div class="form-group">
        <label>Price</label>
        <input class="form-control" name="price"
               [(ngModel)]="product.price"
               required pattern="^[0-9\.]+$" />
```

```
    </div>

    <button type="submit" class="btn btn-primary"
            [class.btn-warning]="editing" [disabled]="form.invalid">
        {{editing ? "Save" : "Create"}}
    </button>
    <button type="reset" class="btn btn-secondary" routerLink="/">Cancel</button>
</form>
```

Значение, присвоенное атрибуту `routerLink`, выбирает маршрут, который выводит таблицу товаров. В листинге 25.12 приведено обновление функционального модуля с шаблоном, который импортирует `RouterModule` — модуль Angular с директивой, выбирающей атрибут `routerLink`.

Листинг 25.12. Добавление директивы маршрутизации в файл core.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule } from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { TableComponent } from "./table.component";
import { FormComponent } from "./form.component";
import { SharedState, SHARED_STATE } from "./sharedState.model";
import { Subject } from "rxjs/Subject";
import { StatePipe } from "./state.pipe";
import { MessageModule } from "../messages/message.module";
import { MessageService } from "../messages/message.service";
import { Message } from "../messages/message.model";
import { Model } from "../model/repository.model";
import { MODES } from "./sharedState.model";
import { RouterModule } from "@angular/router";

@NgModule({
    imports: [BrowserModule, FormsModule, ModelModule, MessageModule,
RouterModule],
    declarations: [TableComponent, FormComponent, StatePipe],
    exports: [ModelModule, TableComponent, FormComponent],
    providers: [{
        provide: SHARED_STATE,
        deps: [MessageService, Model],
        useFactory: (messageService, model) => {
            return new Subject<SharedState>();
        }
    }]
})
export class CoreModule { }
```

Эффект маршрутизации

Когда все изменения будут сохранены, вы сможете осуществлять навигацию в приложении с использованием кнопок Edit, Create New Product и Cancel (рис. 25.3).

Не вся функциональность приложения работает, но сейчас наступил удобный момент для исследования эффекта включения маршрутизации в приложение.

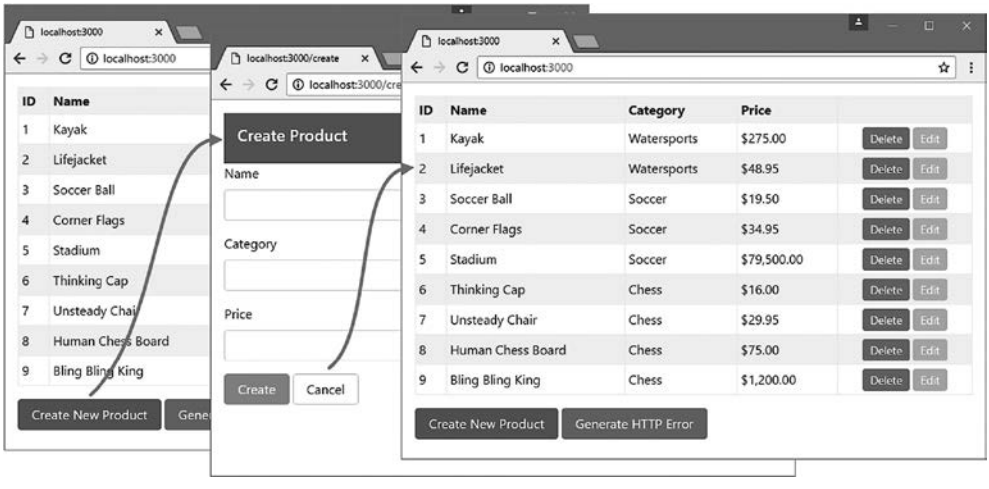


Рис. 25.3. Использование маршрутизации для выполнения навигации в приложении

Введите корневой URL-адрес приложения (*http://localhost:3000*) и щелкните на кнопке Create New Product. Система маршрутизации Angular заменяет URL-адрес, отображаемый в браузере, следующим:

http://localhost:3000/form/create

Если понаблюдать за выводом сервера HTTP для разработки во время перехода, вы заметите, что сервер не получает запросы нового контента. Все изменения происходят исключительно в приложении Angular, и новые запросы HTTP не выдаются.

Новый URL-адрес обрабатывается системой маршрутизации Angular, которая может сопоставить новый URL-адрес с этим маршрутом из файла app.routing.ts.

```
...
{ path: "form/create", component: FormComponent },
...
```

Система маршрутизации учитывает элемент `base` в файле index.html при поиске соответствия URL с маршрутом. Элемент `base` настраивается со значением `href /`, которое объединяется с путем в маршруте при поиске соответствия для URL-адреса `/form/create`.

Свойство `component` сообщает системе маршрутизации Angular, что она должна отобразить компонент `FormComponent`. Система создает новый экземпляр класса `FormComponent`, а контент его шаблона используется как контент элемента `router-outlet` в шаблоне корневого компонента.

Если щелкнуть на кнопке Cancel под формой, процесс повторяется, но на этот раз браузер возвращается к корневому URL-адресу приложения, который сопоставляется с маршрутом, компонент пути которого является пустой строкой:

```
{ path: "", component: TableComponent }
```

Маршрут приказывает Angular отобразить `TableComponent`. Создается новый экземпляр `TableComponent`, а его шаблон используется как контент элемента `router-outlet`, в результате чего данные модели отображаются для пользователя.

В этом состоит суть маршрутизации: URL-адрес браузера изменяется; это заставляет систему маршрутизации свериться с конфигурацией, чтобы определить, какой компонент нужно вывести для пользователя. В этом процессе задействовано множество параметров и возможностей, однако базовый смысл маршрутизации именно таков. Если вы будете помнить об этом, то по крайней мере будете двигаться в правильном направлении.

ОПАСНОСТИ РУЧНОГО ИЗМЕНЕНИЯ URL

Директива routerLink задает URL через вызов JavaScript API, который сообщает браузеру, что изменение касается текущего документа и не требует запроса HTTP к серверу. Если ввести в окне браузера URL-адрес, для которого найдется соответствие в системе маршрутизации, результат будет похож на ожидаемое изменение, но в действительности происходит нечто совершенно иное. Понаблюдайте за выводом сервера HTTP для разработки при ручном вводе следующего URL в браузере:

http://localhost:3000/form/create

Вместо того чтобы обрабатывать это изменение в приложении Angular, браузер отправляет запрос HTTP серверу, который перезагружает приложение. После того как приложение будет загружено, система маршрутизации анализирует URL в браузере, находит соответствие с одним из маршрутов в конфигурации и отображает компонент FormComponent.

Вся эта схема работает потому, что пакет lite-server, который предоставляет функциональность сервера HTTP для разработки, возвращает содержимое файла index.html для URL, не имеющих соответствующих файлов на диске. Например, при запросе следующего URL

http://localhost:3000/this/does/not/exist

браузер выдаст ошибку, потому что запрос предоставляет браузеру содержимое файла index.html, который был использован для загрузки и запуска приложения Angular. Анализируя URL, система маршрутизации не находит подходящего маршрута и выдает ошибку.

Здесь есть два важных обстоятельства. Во-первых, при тестировании конфигурации маршрутизации вы должны следить за запросами HTTP, которые выдает браузер, потому что иногда можно получить правильный результат по неправильным причинам. На быстрой машине вы даже не заметите, что приложение было перезагружено и перезапущено браузером.

Во-вторых, необходимо помнить, что URL должен изменяться директивой routerLink (или одной из аналогичных возможностей, предоставляемых модулем маршрутизации), а не вручную в адресной строке браузера.

Наконец, поскольку пользователи не знают о различиях между программными и ручными изменениями URL, ваша конфигурация маршрутизации должна справляться с URL, не соответствующими маршрутам (см. главу 26).

Завершение реализации маршрутизации

Добавление маршрутизации в приложение — хорошее начало, но многие функциональные возможности приложения еще не работают. Например, при щелчке на кнопке Edit отображается форма, но она не заполнена и в ней не используются цветовые признаки редактирования. В следующих разделах я использую функциональность, предоставляемую системой маршрутизации, и доработаю приложение, чтобы все работало так, как ожидалось.

Обработка изменений маршрутов в компонентах

Компонент формы работает некорректно, потому что он не получил уведомления о том, что пользователь щелкнул на кнопке для редактирования продукта. Проблема возникла из-за того, что система маршрутизации создает новые экземпляры классов компонентов только тогда, когда они ей понадобятся; это означает, что объект `FormComponent` создается только после щелчка на кнопке Edit. Если щелкнуть на кнопке Cancel под формой, а затем снова щелкнуть на кнопке Edit, будет создан второй экземпляр `FormComponent`.

Это создает проблемы временной последовательности при взаимодействии компонентов формы и таблицы через объект Reactive Extensions `Subject`. Объект `Subject` передает события только подписчикам, поступившим после вызова метода `subscribe`. Введение маршрутизации означает, что объект `FormComponent` создается после того, как событие, описывающее операцию редактирования, уже было отправлено.

Проблему можно решить заменой `Subject` на объект `BehaviorSubject`, который отправляет самое последнее событие подписчикам при вызове метода `subscribe`. Другое, более элегантное решение, особенно если вспомнить, что глава посвящена системе маршрутизации, заключается в использовании URL для организации взаимодействия между компонентами.

Angular предоставляет службу, которая может передаваться компонентам для получения подробной информации о текущем маршруте. Отношения между службой и типами, к которым она предоставляет доступ, на первый взгляд кажутся сложными, но все встанет на свои места, если понаблюдать за развитием примеров и некоторыми способами применения маршрутизации.

Класс, от которого компоненты объявляют зависимость, называется `ActivatedRoute`. Для целей этого раздела оно определяет одно важное свойство, описанное в табл. 25.5. Есть и другие свойства, которые будут описаны позже в этой главе, но сейчас они для нас неактуальны.

Таблица 25.5. Свойство ActivatedRoutes

Имя	Описание
snapshot	Свойство возвращает объект ActivatedRouteSnapshot, описывающий текущий маршрут

Свойство `snapshot` возвращает экземпляр класса `ActivatedRouteSnapshot`, который содержит информацию о маршруте, приведшем к отображению текущего компонента. Свойства этого класса описаны в табл. 25.6.

Таблица 25.6. Свойства ActivatedRouteSnapshot

Имя	Описание
url	Свойство возвращает массив объектов UrlSegment, каждый из которых описывает один сегмент URL-адреса, соответствующего текущему маршруту
params	Свойство возвращает объект Params с описанием параметров URL, индексированных по имени
queryParams	Свойство возвращает объект Params с описанием параметров запроса URL, индексированных по имени
fragment	Свойство возвращает объект string, содержащий фрагмент URL

Для нашего примера самым важным является свойство `url`, потому что оно позволяет компоненту проанализировать сегменты текущего URL-адреса и извлечь из них информацию, необходимую для выполнения операции. Свойство `url` возвращает массив объектов `UrlSegment`, свойства которых перечислены в табл. 25.7.

Таблица 25.7. Свойства URLSegment

Имя	Описание
Path	Свойство возвращает строку со значением сегмента
parameters	Свойство возвращает индексированную коллекцию параметров (см. раздел «Использование параметров маршрута»)

Чтобы определить, какой маршрут был активизирован пользователем, компонент формы может объявить зависимость от `ActivatedRoute`, а затем использовать полученный объект для анализа сегментов URL, как показано в листинге 25.13.

Листинг 25.13. Анализ активного маршрута в файле form.component.ts

```
import { Component, Inject } from "@angular/core";
import { NgForm } from "@angular/forms";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
//import { MODES, SharedState, SHARED_STATE } from "./sharedState.model";
//import { Observable } from "rxjs/Observable";
//import "rxjs/add/operator/filter";
//import "rxjs/add/operator/map";
//import "rxjs/add/operator/distinctUntilChanged";
//import "rxjs/add/operator/skipWhile";
import { ActivatedRoute } from "@angular/router";

@Component({
    selector: "paForm",
    moduleId: module.id,
    templateUrl: "form.component.html",
    styleUrls: ["form.component.css"]
```

```
})
export class FormComponent {
    product: Product = new Product();

    constructor(private model: Model, activeRoute: ActivatedRoute) {
        this.editing = activeRoute.snapshot.url[1].path == "edit";
    }

    editing: boolean = false;

    submitForm(form: NgForm) {
        if (form.valid) {
            this.model.saveProduct(this.product);
            this.product = new Product();
            form.reset();
        }
    }

    resetForm() {
        this.product = new Product();
    }
}
```

Компонент уже не использует Reactive Extensions для получения событий. Вместо этого он анализирует второй сегмент URL активного маршрута для задания свойства `editing`, которое определяет режим отображения информации (создание или редактирование товара). Щелкнув на кнопке Edit в таблице, вы увидите правильные цветовые пометки (рис. 25.4).

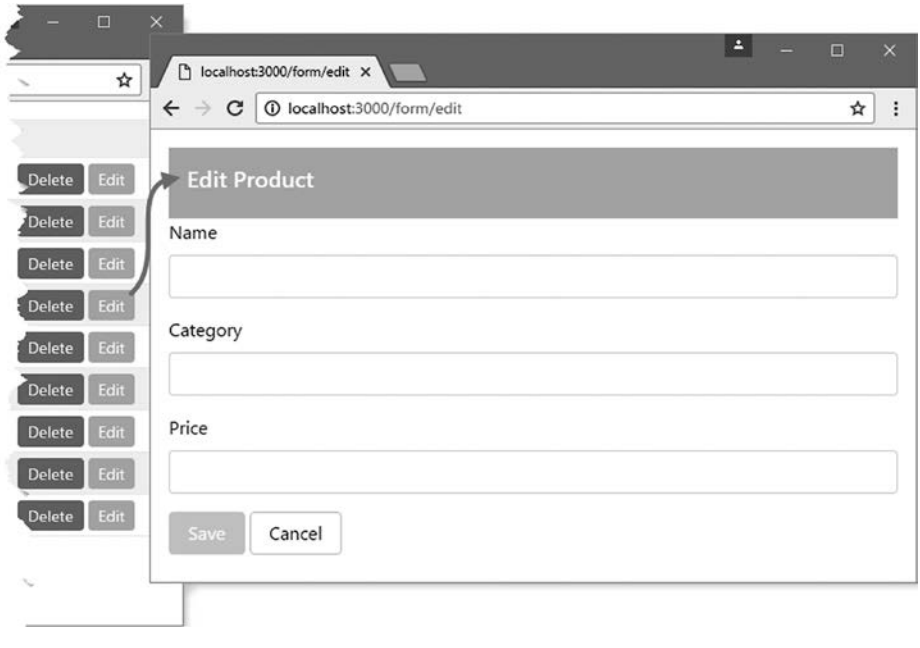


Рис. 25.4. Использование активного маршрута в компоненте

ПРОБЛЕМА ВВОДА URL

Сервер HTTP для разработки, использованный в книге, — lite-server — возвращает содержимое файла index.html при запросе браузером URL-адреса, не имеющего соответствующего файла в проекте. В сочетании с автоматической перезагрузкой браузера это означает, что вы можете заставить приложение запуститься на конкретном URL. Например, если щелкнуть на одной из кнопок Edit в предыдущем примере, браузер перейдет по URL-адресу /form/edit. Если изменить файл, браузер автоматически перезагрузит приложение и запросит у сервера HTTP URL-адрес /form/edit, не соответствующий файлу в папке exampleApp, для которого сервер HTTP ответит содержимым файла index.html.

Данная возможность может быть полезной, потому что она предотвращает выдачу ошибки 404 при переходе приложения по маршруту, который еще не был реализован. Но она также может создать проблемы, потому что приложение не прошло последовательность шагов навигации от корневого URL-адреса, необходимую для создания данных состояния для маршрута, по которому переходит браузер.

В главе 27 я покажу, как использовать защиту маршрута для предотвращения такого поведения. До этого момента некоторые из последующих примеров не будут работать так, как предполагается, если не запросить явно URL-адрес http://localhost:3000, а затем воспользоваться кнопками в макете приложения.

Использование параметров маршрутов

В процессе подготовки конфигурации маршрутизации для приложения мы определили два маршрута, целью которых является компонент формы:

```
...
{ path: "form/edit", component: FormComponent },
{ path: "form/create", component: FormComponent },
...
```

Пытаясь сопоставить маршрут с URL-адресом, Angular проверяет каждый сегмент и смотрит, подходит ли он для URL, по которому был выполнен переход. Оба URL-адреса состоят из *статических сегментов*; это означает, что они должны точно совпадать с URL перехода перед тем, как Angular активизирует маршрут.

Маршруты Angular обладают большей гибкостью: они могут включать *параметры*, которые позволяют любому значению сегмента совпадать с соответствующим сегментом в URL-адресе перехода. Это означает, что маршруты, выбирающие один и тот же компонент с похожими URL-адресами, можно объединить в один маршрут (листинг 25.14).

Листинг 25.14. Консолидация маршрутов в файле app.routing.ts

```
import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "./core/table.component";
import { FormComponent } from "./core/form.component";

const routes: Routes = [
    { path: "form/:mode", component: FormComponent },
```

```
    { path: "", component: TableComponent }
]

export const routing = RouterModule.forRoot(routes);
```

Второй сегмент измененного URL-адреса определяет параметр маршрута, обозначаемый двоеточием (:), за которым следует имя. В данном случае параметр маршрута называется mode. Этот маршрут будет соответствовать любому URL-адресу из двух сегментов, первый из которых содержит form (табл. 25.8). Содержимое второго сегмента будет присвоено параметру с именем mode.

Таблица 25.8. Поиск соответствия URL с параметром маршрута

URL	Результат
http://localhost:3000/form	Соответствия нет — слишком мало сегментов
http://localhost:3000/form/create	Есть соответствие, параметру mode присваивается create
http://localhost:3000/form/london	Есть соответствие, параметру mode присваивается london
http://localhost:3000/product/edit	Соответствия нет — первый сегмент не содержит form
http://localhost:3000/form/edit/1	Соответствия нет — слишком много сегментов

Параметры маршрута упрощают программную обработку маршрутов, потому что значение параметра может быть получено по имени (листинг 25.15).

Листинг 25.15. Чтение параметра маршрута из файла form.component.ts

```
import { Component, Inject } from "@angular/core";
import { NgForm } from "@angular/forms";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
import { ActivatedRoute } from "@angular/router";

@Component({
    selector: "paForm",
    moduleId: module.id,
    templateUrl: "form.component.html",
    styleUrls: ["form.component.css"]
})
export class FormComponent {
    product: Product = new Product();

    constructor(private model: Model, activeRoute: ActivatedRoute) {
        this.editing = activeRoute.snapshot.params["mode"] == "edit";
    }

    // ...методы и свойства опущены для краткости...
}
```

Для получения необходимой информации компоненту необязательно знать структуру URL. Вместо этого он может воспользоваться свойством `params` класса `ActivatedRouteSnapshot`, чтобы получить коллекцию значений параметров, индексированную по имени. Компонент получает значение параметра `mode` и использует его для присваивания значения свойства `editing`.

Множественные параметры маршрутов

Чтобы сообщить компоненту формы, какой товар был выбран кнопкой Edit, нам понадобится второй параметр маршрута. Так как Angular ищет соответствия URL на основании количества содержащихся в них сегментов, это означает, что маршруты для компонента формы придется разбить заново (листинг 25.16). Цикл объединения и последующего расширения маршрутов типичен для многих проектов разработки: с увеличением количества информации, включаемой в маршрутные URL-адреса, расширяется функциональность приложения.

Листинг 25.16. Добавление маршрута в файл app.routing.ts

```
import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "./core/table.component";
import { FormComponent } from "./core/form.component";

const routes: Routes = [
    { path: "form/:mode/:id", component: FormComponent },
    { path: "form/:mode", component: FormComponent },
    { path: "", component: TableComponent }]

export const routing = RouterModule.forRoot(routes);
```

Новый маршрут будет соответствовать любому URL-адресу из трех сегментов, первым из которых является сегмент `form`. Чтобы создать URL-адреса для этого маршрута, необходимо иначе строить выражения `routerLink` в шаблоне, потому что третий сегмент должен генерироваться динамически для каждой кнопки Edit в таблице товаров (листинг 25.17).

Листинг 25.17. Генерирование динамических URL в файле table.component.html

```
<table class="table table-sm table-bordered table-striped">
    <tr>
        <th>ID</th><th>Name</th><th>Category</th><th>Price</th><th></th>
    </tr>
    <tr *ngFor="let item of getProducts()">
        <td style="vertical-align:middle">{{item.id}}</td>
        <td style="vertical-align:middle">{{item.name}}</td>
        <td style="vertical-align:middle">{{item.category}}</td>
        <td style="vertical-align:middle">
            {{item.price | currency:"USD":true }}
        </td>
        <td class="text-xs-center">
            <button class="btn btn-danger btn-sm" (click)="deleteProduct(item.id)">
                Delete
            </button>
```

```
            <button class="btn btn-warning btn-sm" (click)="editProduct(item.id)"
                    [routerLink]="['/form', 'edit', item.id]">
                Edit
            </button>
        </td>
    </tr>
</table>
<button class="btn btn-primary" (click)="createProduct()" routerLink="/form/
create">
    Create New Product
</button>
<button class="btn btn-danger" (click)="deleteProduct(-1)">
    Generate HTTP Error
</button>
```

Атрибут `routerLink` теперь заключается в квадратные скобки; тем самым вы сообщаете Angular, что значение атрибута должно использоваться как выражение привязки данных. Выражение задается в виде массива, в котором каждый элемент содержит значение одного сегмента. Первые два сегмента — литералы, которые будут включены в целевой URL-адрес без изменений. Третий сегмент обрабатывается для включения значения свойства `id` текущего объекта `Product`, обрабатываемого директивой `ngIf`, как и в других выражениях шаблона. Директива `routerLink` объединяет отдельные сегменты и строит из них URL вида `/form/edit/2`.

В листинге 25.18 показано, как компонент формы получает значение нового параметра маршрута и использует его для выбора редактируемого товара.

Листинг 25.18. Использование нового параметра маршрута в файле form.component.ts

```
import { Component, Inject } from "@angular/core";
import { NgForm } from "@angular/forms";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
import { ActivatedRoute } from "@angular/router";

@Component({
    selector: "paForm",
    moduleId: module.id,
    templateUrl: "form.component.html",
    styleUrls: ["form.component.css"]
})
export class FormComponent {
    product: Product = new Product();

    constructor(private model: Model, activeRoute: ActivatedRoute) {
        this.editing = activeRoute.snapshot.params["mode"] == "edit";
        let id = activeRoute.snapshot.params["id"];
        if (id != null) {
            Object.assign(this.product, model.getProduct(id) || new Product());
        }
    }

    // ...методы и свойства опущены для краткости...
}
```

Когда пользователь щелкает на кнопке Edit, активизируемый URL-адрес сообщает компоненту формы, что запрошена операция редактирования и что будет изменяться такой-то товар. Это позволило правильно заполнить форму (рис. 25.5).

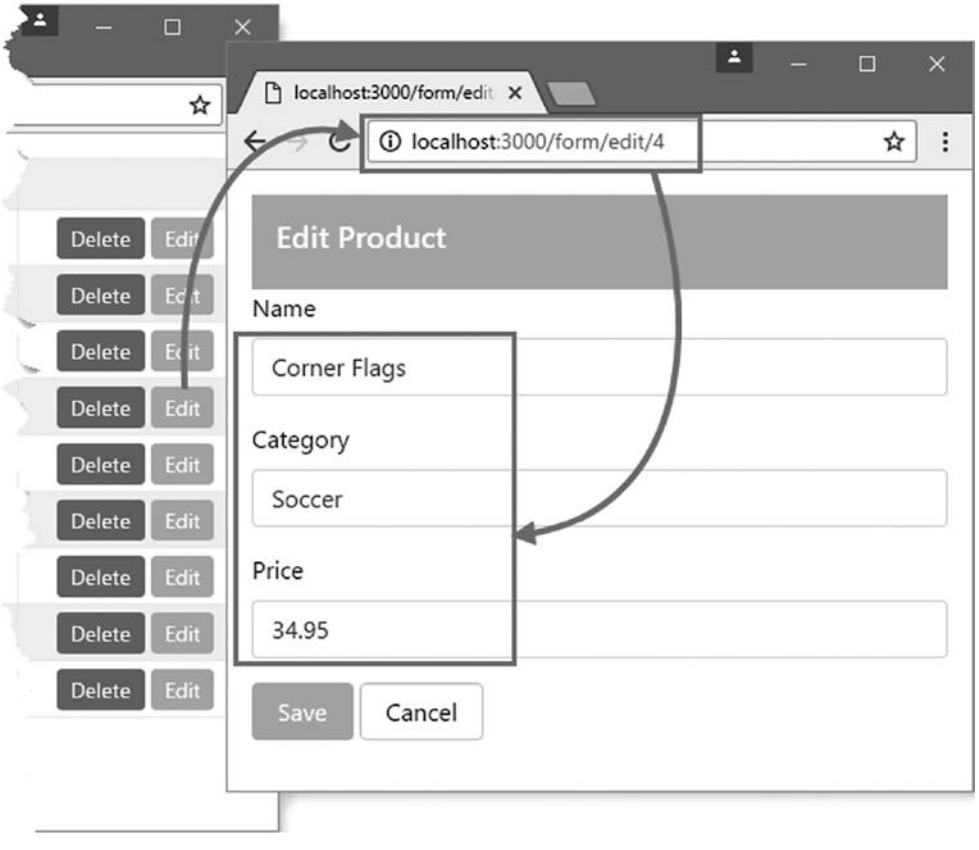


Рис. 25.5. Использование сегментов URL для передачи информации

ПРИМЕЧАНИЕ

Обратите внимание: в листинге 25.18 мне приходится проверять, что объект Product был успешно прочитан из модели данных, а если этого не произошло — создавать новый объект. Это важный момент: ведь данные в модели читаются асинхронно и могут еще не поступить к моменту отображения компонента формы, если пользователь запросил URL напрямую. Также могут возникнуть проблемы при разработке, если изменения в коде приложения инициируют перекомпиляцию с последующей перезагрузкой URL-адреса, открытого до внесения изменений. Это приведет к ошибке, когда Angular попытается напрямую перейти по маршруту (как предполагалось, это станет ненужным после заполнения модели данных). В главе 27 я объясню, как остановить активизацию маршрутов до выполнения некоторого условия (например, поступления данных).

Необязательные параметры маршрутов

Необязательные параметры маршрутов позволяют URL включать дополнительную информацию для передачи остальным частям приложения, не являющуюся абсолютно необходимой для приложения.

Параметры маршрутизации этого типа выражаются с использованием *матричной записи* URL, которая не является частью спецификации URL, но при этом поддерживается браузерами. Пример URL с необязательными параметрами маршрутов:

http://localhost:3000/form/edit/2;name=Lifejacket;price=48.95

Необязательные параметры маршрутизации разделяются символом «точка с запятой» (`;`), и этот URL включает необязательные параметры `name` и `price`.

Для демонстрации использования необязательных параметров в листинге 25.19 добавляется необязательный параметр маршрута, который включает редактируемый объект как часть URL. Эта информация некритична, потому что компонент формы может получить данные из модели, но получение данных через URL маршрутизации избавит от лишней работы.

Листинг 25.19. Использование необязательного параметра маршрута в файле table.component.html

```
<table class="table table-sm table-bordered table-striped">
    <tr>
        <th>ID</th><th>Name</th><th>Category</th><th>Price</th><th></th>
    </tr>
    <tr *ngFor="let item of getProducts()">
        <td style="vertical-align:middle">{{item.id}}</td>
        <td style="vertical-align:middle">{{item.name}}</td>
        <td style="vertical-align:middle">{{item.category}}</td>
        <td style="vertical-align:middle">
            {{item.price | currency:"USD":true }}
        </td>
        <td class="text-xs-center">
            <button class="btn btn-danger btn-sm" (click)="deleteProduct(item.id)">
                Delete
            </button>
            <button class="btn btn-warning btn-sm" (click)="editProduct(item.id)"
                [routerLink]="['/form', 'edit', item.id,
                    {name: item.name, category: item.category, price: item.price}]">
                Edit
            </button>
        </td>
    </tr>
</table>
<button class="btn btn-primary" (click)="createProduct()" routerLink="/form/create">
    Create New Product
</button>
<button class="btn btn-danger" (click)="deleteProduct(-1)">
    Generate HTTP Error
</button>
```

Необязательные значения выражаются в виде объектных литералов, имена свойств которых определяют необязательный параметр. В данном примере это свойства `name`, `category` и `price`, а их значения задаются с использованием объекта, обрабатываемого директивой `ngIf`. Необязательные параметры создают URL следующего вида:

http://localhost:3000/form/edit/5;name=Stadium;category=Soccer;price=79500

В листинге 25.20 показано, как компонент формы проверяет присутствие необязательных параметров. Если они были включены в URL, то значения параметров используются для предотвращения запроса к модели данных.

Листинг 25.20. Получение необязательных параметров маршрута в файле form.component.ts

```
...
constructor(private model: Model, activeRoute: ActivatedRoute) {
    this.editing = activeRoute.snapshot.params["mode"] == "edit";
    let id = activeRoute.snapshot.params["id"];
    if (id != null) {
        let name = activeRoute.snapshot.params["name"];
        let category = activeRoute.snapshot.params["category"];
        let price = activeRoute.snapshot.params["price"];

        if (name != null && category != null && price != null) {
            this.product.id = id;
            this.product.name = name;
            this.product.category = category;
            this.product.price = Number.parseFloat(price);
        } else {
            Object.assign(this.product, model.getProduct(id) || new Product());
        }
    }
}
...
```

Вы обращаетесь к необязательным параметрам маршрутов так же, как к обязательным параметрам. Компонент обязан проверить их присутствие и продолжить работу, если они не входят в URL. В данном случае компонент возвращается к выборке из модели данных, если URL не содержит нужных необязательных параметров.

Навигация в программном коде

Использование атрибута `routerLink` упрощает настройку навигации в шаблонах, но приложению часто требуется инициировать навигацию по поручению пользователя в компоненте или директиве.

Чтобы предоставить доступ к системе маршрутизации из структурных блоков (таких, как директивы и компоненты), Angular предоставляет класс `Router`, который доступен в виде службы через механизм внедрения зависимостей. Самые полезные методы и свойства этого класса описаны в табл. 25.9.

Таблица 25.9. Некоторые методы и свойства Router

Имя	Описание
navigated	Свойство типа boolean возвращает true, если было хотя бы одно событие навигации, и false в противном случае
url	Свойство возвращает активный URL-адрес
isActive(url, exact)	Метод возвращает true, если заданный URL совпадает с URL, определенным активным маршрутом. Аргумент exact указывает, должны ли все сегменты заданного URL совпадать с текущим, чтобы метод вернул true
events	Свойство возвращает объект Observable<Event>, который может использоваться для отслеживания навигационных изменений. За подробностями обращайтесь к разделу «Получение событий навигации»
navigateByUrl(url, extras)	Метод выполняет навигацию к заданному URL-адресу. Результатом метода является объект Promise, который разрешается в true, если навигация прошла успешно, false — в противном случае, или отклоняется при возникновении ошибки
navigate(commands, extras)	Метод выполняет навигацию по массиву сегментов. Объект extras может использоваться для определения того, задается ли URL относительно текущего маршрута. Результатом метода является объект Promise, который разрешается в true, если навигация прошла успешно, false — в противном случае, или отклоняется при возникновении ошибки

Методы `navigate` и `navigateByUrl` позволяют легко выполнить навигацию внутри структурного блока (например, компонента). В листинге 25.21 продемонстрировано применение `Router` в компоненте формы для перенаправления приложения обратно к таблице после создания или обновления товара.

Листинг 25.21. Программная навигация в файле form.component.ts

```
import { Component, Inject } from "@angular/core";
import { NgForm } from "@angular/forms";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
import { ActivatedRoute, Router } from "@angular/router";

@Component({
    selector: "paForm",
    moduleId: module.id,
    templateUrl: "form.component.html",
    styleUrls: ["form.component.css"]
})

export class FormComponent {
    product: Product = new Product();

    constructor(private model: Model, activeRoute: ActivatedRoute,
```

```
                private router: Router) {

        this.editing = activeRoute.snapshot.params["mode"] == "edit";
        let id = activeRoute.snapshot.params["id"];
        if (id != null) {
            let name = activeRoute.snapshot.params["name"];
            let category = activeRoute.snapshot.params["category"];
            let price = activeRoute.snapshot.params["price"];

            if (name != null && category != null && price != null) {
                this.product.id = id;
                this.product.name = name;
                this.product.category = category;
                this.product.price = Number.parseFloat(price);
            } else {
                Object.assign(this.product, model.getProduct(id) || new Product());
            }
        }
    }

    editing: boolean = false;

    submitForm(form: NgForm) {
        if (form.valid) {
            this.model.saveProduct(this.product);
            //this.product = new Product();
            //form.reset();
            this.router.navigateByUrl("/");
        }
    }

    resetForm() {
        this.product = new Product();
    }
}
```

Компонент получает объект `Router` в аргументе конструктора и использует его в методе `submitForm` для возврата к корневому URL-адресу приложения. Две команды, которые были закомментированы в методе `submitForm`, больше не нужны, потому что система маршрутизации уничтожит компонент формы после того, как он перестанет отображаться, что означает, что сброс состояния формы не нужен.

В результате щелчок на кнопке Save или Create на форме приведет к тому, что приложение отобразит таблицу товаров (рис. 25.6).

Получение событий навигации

Во многих приложениях встречаются компоненты или директивы, которые не задействованы напрямую в навигации приложения, но при этом все равно должны знать о выполнении навигации. Пример такого рода встречается в компоненте сообщений, который выводит информационные сообщения и ошибки для пользователя. Этот компонент всегда выводит последнее сообщение, даже если информация устарела и вряд ли пригодится пользователю. Чтобы увидеть суть проблемы,

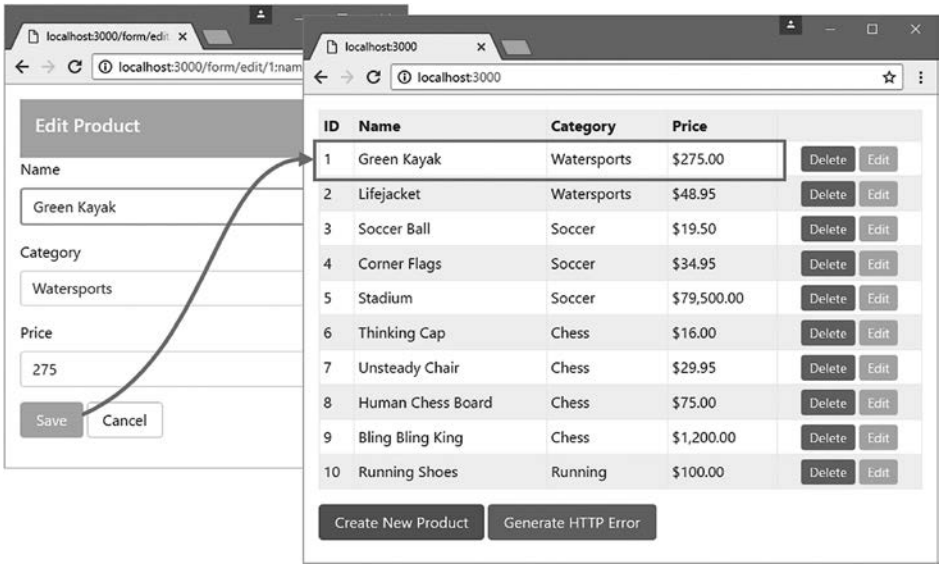


Рис. 25.6. Программная навигация

щелкните на кнопке Generate HTTP Error, а затем на кнопке Create New Product или на одной из кнопок Edit; сообщение об ошибке продолжает выводиться, даже если пользователь перешел к другой части приложения.

Свойство `events`, определяемое классом `Router`, возвращает объект `Observable<Event>`, который выдает последовательность объектов `Event`, описывающих изменения в системе маршрутизации. Всего существуют пять типов событий, предоставляемых через объект `Observer` (табл. 25.10).

Таблица 25.10. Типы событий, предоставляемых Observer для Router.events

Имя	Описание
NavigationStart	Событие отправляется в начале процесса навигации
RoutesRecognized	Событие отправляется, когда система маршрутизации сопоставляет URL-адрес с маршрутом
NavigationEnd	Событие отправляется при успешном завершении процесса навигации
NavigationError	Событие отправляется, если в процессе навигации происходит ошибка
NavigationCancel	Событие отправляется при отмене процесса навигации

Все классы событий определяют свойство `id`, которое возвращает число, увеличивающееся при каждой навигации, и свойство `url`, которое возвращает целевой URL-адрес. События `RoutesRecognized` и `NavigationEnd` также определяют свойство `urlAfterRedirects`, которое возвращает URL, по которому была выполнена навигация.

Для решения проблемы с системой маршрутизации листинг 25.22 подписывается на объект `Observer`, предоставляемый свойством `Router.events`, и очищает сооб-

щение, выводимое для пользователя, при получении события `NavigationEnd` или `NavigationCancel`.

Листинг 25.22. Реакция на события навигации в файле message.component.ts

```
import { Component } from "@angular/core";
import { MessageService } from "./message.service";
import { Message } from "./message.model";
import { Observable } from "rxjs/Observable";
import { Router, NavigationEnd, NavigationCancel } from "@angular/router";
import "rxjs/add/operator/filter";

@Component({
    selector: "paMessages",
    moduleId: module.id,
    templateUrl: "message.component.html",
})

export class MessageComponent {
    lastMessage: Message;

    constructor(messageService: MessageService, router: Router) {
        messageService.messages.subscribe(m => this.lastMessage = m);
        router.events
            .filter(e => e instanceof NavigationEnd || e instanceof
                         NavigationCancel)
            .subscribe(e => { this.lastMessage = null; });
    }
}
```

Метод `filter` используется для выбора одного типа события, а метод `subscribe` обновляет свойство `lastMessage`, что приводит к стиранию сообщения, отображаемого компонентом. Листинг 25.23 импортирует функциональность маршрутизации в модуль сообщений. (В принципе, это не обязательно для работы приложения, так как функциональность маршрутизации уже импортируется корневым модулем, но импортирование всей функциональности, необходимой для каждого модуля, — полезная привычка.)

Листинг 25.23. Импортирование модуля маршрутизации в файле message.module.ts

```
import { NgModule, ErrorHandler } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { MessageComponent } from "./message.component";
import { MessageService } from "./message.service";
import { MessageErrorHandler } from "./errorHandler";
import { RouterModule } from "@angular/router";

@NgModule({
    imports: [BrowserModule, RouterModule],
    declarations: [MessageComponent],
    exports: [MessageComponent],
    providers: [MessageService,
        { provide: ErrorHandler, useClass: MessageErrorHandler }]
})
export class MessageModule { }
```

В результате этих изменений сообщения выводятся только до следующего события навигации (рис. 25.7).

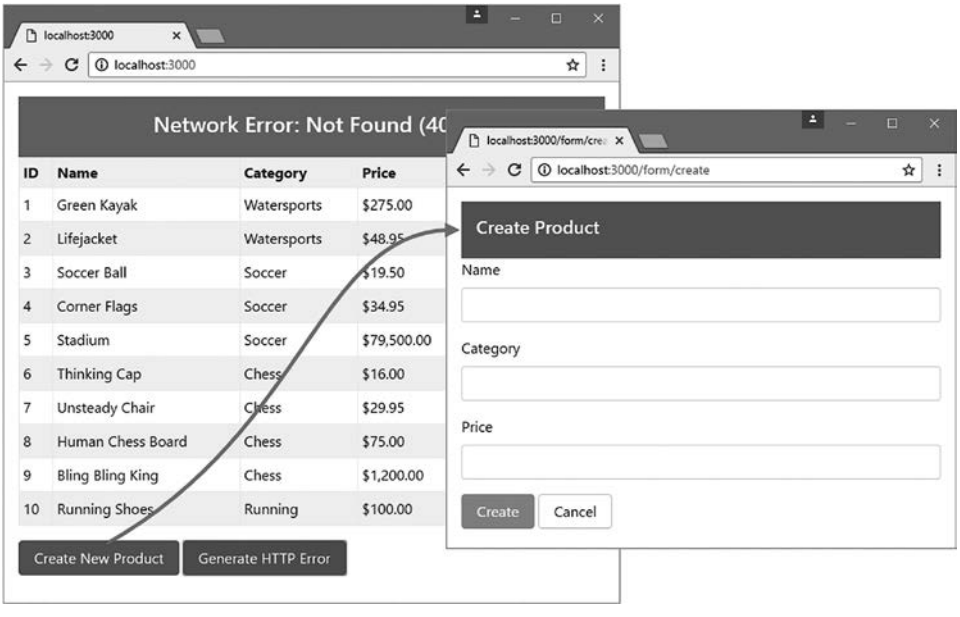


Рис. 25.7. Реакция на события навигации

Удаление привязок событий и вспомогательного кода

Одно из преимуществ системы маршрутизации заключается в том, что она может упростить приложения за счет замены привязок событий и вызываемых ими методов изменения в навигации. Чтобы завершить реализацию маршрутизации, остается удалить последние следы предыдущего механизма, который использовался для координации взаимодействия компонентов. В листинге 25.24 в шаблоне компонента таблицы закомментированы привязки событий, которые использовались для реакции на нажатие пользователем кнопки Create New Product или Edit. (Привязка события для кнопок Delete по-прежнему необходима.)

Листинг 25.24. Удаление привязки событий в файле table.component.html

```
<table class="table table-sm table-bordered table-striped">
    <tr>
        <th>ID</th><th>Name</th><th>Category</th><th>Price</th><th></th>
    </tr>
    <tr *ngFor="let item of getProducts()">
        <td style="vertical-align:middle">{{item.id}}</td>
        <td style="vertical-align:middle">{{item.name}}</td>
        <td style="vertical-align:middle">{{item.category}}</td>
        <td style="vertical-align:middle">
            {{item.price | currency:"USD":true }}
```

```
        </td>
        <td class="text-xs-center">
            <button class="btn btn-danger btn-sm" (click)="deleteProduct(item.id)">
                Delete
            </button>
            <button class="btn btn-warning btn-sm"
                [routerLink]="['/form', 'edit', item.id,
                    {name: item.name, category: item.category, price: item.price}]">
                Edit
            </button>
        </td>
    </tr>
</table>
<button class="btn btn-primary" routerLink="/form/create">
    Create New Product
</button>
<button class="btn btn-danger" (click)="deleteProduct(-1)">
    Generate HTTP Error
</button>
```

В листинге 25.25 показаны соответствующие изменения в компоненте: из него удаляются методы, вызывавшиеся привязками событий, и зависимости от службы, которая сигнализировала о редактировании или создании товара.

Листинг 25.25. Удаление кода обработки событий в файле table.component.ts

```
import { Component, Inject } from "@angular/core";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
//import { MODES, SharedState, SHARED_STATE } from "./sharedState.model";
//import { Observer } from "rxjs/Observer";

@Component({
    selector: "paTable",
    moduleId: module.id,
    templateUrl: "table.component.html"
})
export class TableComponent {

    constructor(private model: Model,
        /*@Inject(SHARED_STATE) private observer: Observer<SharedState>*/) { }

    getProduct(key: number): Product {
        return this.model.getProduct(key);
    }

    getProducts(): Product[] {
        return this.model.getProducts();
    }

    deleteProduct(key: number) {
        this.model.deleteProduct(key);
    }

    //editProduct(key: number) {
    //    this.observer.next(new SharedState(MODES.EDIT, key));
    //}
```

```
    //createProduct() {
    //    this.observer.next(new SharedState(MODES.CREATE));
    //}
}
```

Служба, используемая для координации компонентов, уже не нужна; в листинге 25.26 она исключается из базового модуля.

Листинг 25.26. Удаление службы общего состояния в файле core.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule } from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { TableComponent } from "./table.component";
import { FormComponent } from "./form.component";
//import { SharedState, SHARED_STATE } from "./sharedState.model";
import { Subject } from "rxjs/Subject";
import { StatePipe } from "./state.pipe";
import { MessageModule } from "../messages/message.module";
import { MessageService } from "../messages/message.service";
import { Message } from "../messages/message.model";
import { Model } from "../model/repository.model";
//import { MODES } from "./sharedState.model";
import { RouterModule } from "@angular/router";

@NgModule({
    imports: [BrowserModule, FormsModule, ModelModule, MessageModule,
RouterModule],
    declarations: [TableComponent, FormComponent, StatePipe],
    exports: [ModelModule, TableComponent, FormComponent],
    //providers: [{
    //    provide: SHARED_STATE,
    //    deps: [MessageService, Model],
    //    useFactory: (messageService, model) => {
    //        return new Subject<SharedState>();
    //    }
    //}]
})
export class CoreModule { }
```

В результате координация между компонентами таблицы и формы реализуется полностью через систему маршрутизации, которая теперь отвечает за отображение компонентов и управление навигацией между ними.

Итоги

В этой главе вы познакомились с системой маршрутизации Angular и узнали, как на основании URL-адреса в приложении выбрать отображаемый контент. Я показал, как создать навигационные ссылки в шаблонах, как выполнять навигацию в компоненте или директиве и как реагировать на навигационные изменения в программном коде. В следующей главе описание системы маршрутизации Angular будет продолжено.

26 Маршрутизация и навигация: часть 2

В предыдущей главе я рассказал о системе маршрутизации URL в Angular и объяснил, как использовать ее для управления отображаемыми компонентами. Система маршрутизации обладает множеством возможностей, описание которых будет продолжено в этой главе и в главе 27. В этой главе основное внимание уделяется созданию более сложных маршрутов, включению маршрутов, которые будут соответствовать любым URL, маршрутам для перенаправления браузера на другие URL, маршрутам для выполнения навигации внутри компонентов и маршрутам для выбора нескольких компонентов.

В табл. 26.1 приведена краткая сводка материала главы.

Таблица 26.1. Сводка материала главы

Проблема	Решение	Листинг
Сопоставление нескольких URL в одном маршруте	Используйте универсальные маршруты	1–10
Перенаправление на другой URL-адрес	Используйте перенаправляющий маршрут	11
Навигация внутри компонента	Используйте относительный URL-адрес	12
Получение уведомлений об изменении активизированного URL-адреса	Используйте объекты Observable, предоставляемые классом ActivatedRoute	13
Стилевое оформление элемента, когда активен конкретный маршрут	Используйте атрибут routerLinkActive	14–17
Использование системы маршрутизации для отображения вложенных компонентов	Определите дочерние маршруты и используйте атрибут router-outlet	18–21

Подготовка проекта

В этой главе мы продолжим использовать проект exampleApp, который был создан в главе 22 и изменялся во всех последующих главах. Для этой главы в класс репозитория следует добавить два метода (листинг 26.1).

ПРИМЕЧАНИЕ

Для работы примеров этой главы необходим пакет Reactive Extensions, созданный в главе 22. Если вы не хотите создавать проект самостоятельно, его вместе с модулем Reactive Extensions можно загрузить в составе бесплатно распространяемого архива исходного кода с сайта apress.com.

Листинг 26.1. Добавление методов в файл repository.model.ts

```
import { Injectable } from "@angular/core";
import { Product } from "./product.model";
import { Observable } from "rxjs/Observable";
import { RestDataSource } from "./rest.datasource";

@Injectable()
export class Model {
    private products: Product[] = new Array<Product>();
    private locator = (p: Product, id: number) => p.id == id;

    constructor(private dataSource: RestDataSource) {
        this.dataSource.getData().subscribe(data => this.products = data);
    }

    getProducts(): Product[] {
        return this.products;
    }

    getProduct(id: number): Product {
        return this.products.find(p => this.locator(p, id));
    }

    getNextProductId(id: number): number {
        let index = this.products.findIndex(p => this.locator(p, id));
        if (index > -1) {
            return this.products[this.products.length > index + 2
                ? index + 1 : 0].id;
        } else {
            return id || 0;
        }
    }

    getPreviousProductid(id: number): number {
        let index = this.products.findIndex(p => this.locator(p, id));
        if (index > -1) {
            return this.products[index > 0
                ? index - 1 : this.products.length - 1].id;
        } else {
            return id || 0;
        }
    }

    saveProduct(product: Product) {
        if (product.id == 0 || product.id == null) {
            this.dataSource.saveProduct(product)
                .subscribe(p => this.products.push(p));
        } else {
```

```
            this.dataSource.updateProduct(product).subscribe(p => {
                let index = this.products
                    .findIndex(item => this.locator(item, p.id));
                this.products.splice(index, 1, p);
            });
        }
    }

    deleteProduct(id: number) {
        this.dataSource.deleteProduct(id).subscribe(() => {
            let index = this.products.findIndex(p => this.locator(p, id));
            if (index > -1) {
                this.products.splice(index, 1);
            }
        });
    }
}
```

Новые методы получают идентификатор, находят соответствующий товар, после чего возвращают идентификаторы следующего и предыдущего объектов в массиве, который используется репозиторием для сбора объектов модели данных. Мы задействуем эту возможность позже в этой главе, когда займемся страничным перебором набора объектов в модели данных.

Для упрощения примера в листинге 26.2 из компонента формы исключаются команды для получения информации о редактируемом товаре с использованием необязательных параметров маршрута.

Листинг 26.2. Исключение необязательных параметров маршрута в файле form.component.ts

```
import { Component, Inject } from "@angular/core";
import { NgForm } from "@angular/forms";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
import { ActivatedRoute, Router } from "@angular/router";

@Component({
    selector: "paForm",
    moduleId: module.id,
    templateUrl: "form.component.html",
    styleUrls: ["form.component.css"]
})
export class FormComponent {
    product: Product = new Product();

    constructor(private model: Model, activeRoute: ActivatedRoute,
        private router: Router) {

        this.editing = activeRoute.snapshot.params["mode"] == "edit";
        let id = activeRoute.snapshot.params["id"];
        if (id != null) {
            Object.assign(this.product, model.getProduct(id) || new Product());
        }
    }
```

```
    editing: boolean = false;

    submitForm(form: NgForm) {
        if (form.valid) {
            this.model.saveProduct(this.product);
            this.router.navigateByUrl("/");
        }
    }
    resetForm() {
        this.product = new Product();
    }
}
```

В листинге 26.3 необязательные параметры исключаются из шаблона компонента таблицы, чтобы они не включались в навигационные URL-адреса кнопок Edit.

Листинг 26.3. Удаление необязательных параметров маршрутов из файла

```
<table class="table table-sm table-bordered table-striped">
    <tr>
        <th>ID</th><th>Name</th><th>Category</th><th>Price</th><th></th>
    </tr>
    <tr *ngFor="let item of getProducts()">
        <td style="vertical-align:middle">{{item.id}}</td>
        <td style="vertical-align:middle">{{item.name}}</td>
        <td style="vertical-align:middle">{{item.category}}</td>
        <td style="vertical-align:middle">
            {{item.price | currency:"USD":true }}
        </td>
        <td class="text-xs-center">
            <button class="btn btn-danger btn-sm" (click)="deleteProduct(item.id)">
                Delete
            </button>
            <button class="btn btn-warning btn-sm"
                [routerLink]="['/form', 'edit', item.id]">
                Edit
            </button>
        </td>
    </tr>
</table>
<button class="btn btn-primary" routerLink="/form/create">
    Create New Product
</button>
<button class="btn btn-danger" (click)="deleteProduct(-1)">
    Generate HTTP Error
</button>
```

Добавление компонентов в проект

Чтобы продемонстрировать некоторые возможности, описанные в этой главе, необходимо добавить компоненты в приложение. Эти компоненты просты, потому что наше внимание сейчас сосредоточено на системе маршрутизации —

вместо того, чтобы добавлять полезные функции в приложение. Создайте файл productCount.component.ts в папке exampleApp/app/core и включите в него определение компонента из листинга 26.4.

ПРИМЕЧАНИЕ

Вы можете опустить атрибут selector в декораторе @Component, если компонент будет отображаться только через систему маршрутизации. Я обычно добавляю его в любом случае, чтобы компонент также можно было применять с использованием элемента HTML.

Листинг 26.4. Файл productCount.component.ts в папке exampleApp/app/core

```
import { Component, KeyValueDiffer,
         KeyValueDiffers, ChangeDetectorRef } from "@angular/core";
import { Model } from "../model/repository.model";

@Component({
    selector: "paProductCount",
    template: `<div class="bg-info p-a-1">There are {{count}} products</div>`
})
export class ProductCountComponent {
    private differ: KeyValueDiffer;
    count: number = 0;

    constructor(private model: Model,
        private keyValueDiffers: KeyValueDiffers,
        private changeDetector: ChangeDetectorRef) {}

    ngOnInit() {
        this.differ = this.keyValueDiffers
            .find(this.model.getProducts())
            .create(this.changeDetector);
    }

    ngDoCheck() {
        if (this.differ.diff(this.model.getProducts()) != null) {
            this.updateCount();
        }
    }

    private updateCount() {
        this.count = this.model.getProducts().length;
    }
}
```

Компонент использует встроенный шаблон для отображения количества товаров в модели данных, которое обновляется при изменении модели данных. Далее создайте файл categoryCount.component.ts в папке exampleApp/app/core и включите определение компонента из листинга 26.5.

Листинг 26.5. Файл categoryCount.component.ts в папке exampleApp/app/core

```
import { Component, KeyValueDiffer,
         KeyValueDiffers, ChangeDetectorRef } from "@angular/core";
import { Model } from "../model/repository.model";

@Component({
    selector: "paCategoryCount",
    template: `<div class="bg-primary p-a-1">There are {{count}} categories</div>`
})
export class CategoryCountComponent {
    private differ: KeyValueDiffer;
    count: number = 0;

    constructor(private model: Model,
        private keyValueDiffers: KeyValueDiffers,
        private changeDetector: ChangeDetectorRef) { }

    ngOnInit() {
        this.differ = this.keyValueDiffers
            .find(this.model.getProducts())
            .create(this.changeDetector);
    }

    ngDoCheck() {
        if (this.differ.diff(this.model.getProducts()) != null) {
            this.count = this.model.getProducts()
                .map(p => p.category)
                .filter((category, index, array) => array.indexOf(category) ==
                                                    index)
                .length;
        }
    }
}
```

Компонент использует диффер для отслеживания изменений в модели данных и подсчета количества уникальных категорий, которое отображается в простом встроенном шаблоне. Для последнего компонента создайте файл notFound.component.ts в папке exampleApp/app/core и включите в него определение компонента из листинга 26.6.

Листинг 26.6. Файл notFound.component.ts в папке exampleApp/app/core

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

@Component({
    selector: "paNotFound",
    template: `<h3 class="bg-danger p-a-1">Sorry, something went wrong</h3>
               <button class="btn btn-primary" routerLink="/">Start Over</button>`
})
export class NotFoundComponent {}
```

Компонент выводит статическое сообщение, которое должно отображаться при возникновении проблем с системой маршрутизации. В листинге 26.7 в базовый модуль добавляются новые компоненты.

Листинг 26.7. Объявление компонентов в файле core.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule } from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { TableComponent } from "./table.component";
import { FormComponent } from "./form.component";
import { Subject } from "rxjs/Subject";
import { StatePipe } from "./state.pipe";
import { MessageModule } from "../messages/message.module";
import { MessageService } from "../messages/message.service";
import { Message } from "../messages/message.model";
import { Model } from "../model/repository.model";
import { RouterModule } from "@angular/router";
import { ProductCountComponent } from "./productCount.component";
import { CategoryCountComponent } from "./categoryCount.component";
import { NotFoundComponent } from "./notFound.component";

@NgModule({
    imports: [BrowserModule, FormsModule, ModelModule, MessageModule,
              RouterModule],
    declarations: [TableComponent, FormComponent, StatePipe,
        ProductCountComponent, CategoryCountComponent, NotFoundComponent],
    exports: [ModelModule, TableComponent, FormComponent]
})
export class CoreModule { }
```

Сохраните изменения и выполните следующую команду из папки exampleApp, чтобы запустить компилятор TypeScript, сервер HTTP для разработки и REST-совместимой веб-службы:

```
npm start
```

Открывается новое окно (или вкладка) браузера с контентом, показанным на рис. 26.1.

Универсальные маршруты и перенаправления

Конфигурация маршрутизации в приложении быстро усложняется, в ней появляется избыточность и всевозможные артефакты, обусловленные структурой приложения. Angular предоставляет два полезных инструмента, которые помогают упростить маршруты и решать возникающие проблемы (см. следующие разделы).

Универсальные маршруты

Система маршрутизации Angular поддерживает специальный путь, обозначаемый двумя звездочками (**), который позволяет маршруту соответствовать любому URL. Основное применение универсального пути — обработка навигации, которая в противном случае вызвала бы ошибку маршрутизации. В листинге 26.8 в шаблон компонента таблицы добавляется кнопка для перехода к маршруту, который не был определен в конфигурации маршрутизации приложения.

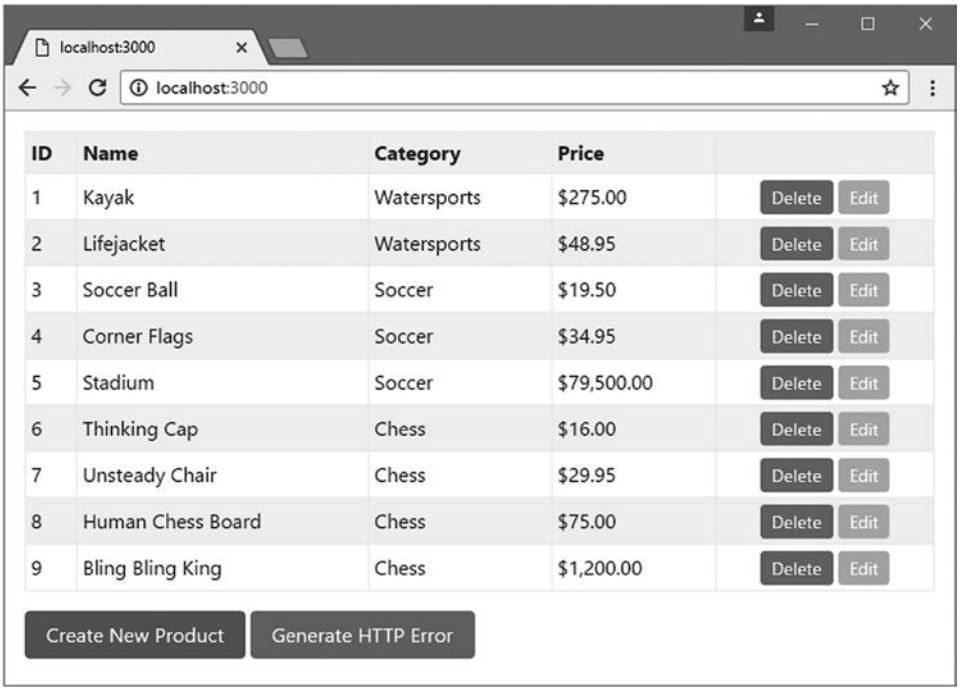


Рис. 26.1. Запуск приложения

Листинг 26.8. Добавление кнопки, генерирующей ошибку, в файл table.component.html

```
<table class="table table-sm table-bordered table-striped">
    <tr>
        <th>ID</th><th>Name</th><th>Category</th><th>Price</th><th></th>
    </tr>
    <tr *ngFor="let item of getProducts()">
        <td style="vertical-align:middle">{{item.id}}</td>
        <td style="vertical-align:middle">{{item.name}}</td>
        <td style="vertical-align:middle">{{item.category}}</td>
        <td style="vertical-align:middle">
            {{item.price | currency:"USD":true }}
        </td>
        <td class="text-xs-center">
            <button class="btn btn-danger btn-sm" (click)="deleteProduct(item.id)">
                Delete
            </button>
            <button class="btn btn-warning btn-sm"
                [routerLink]="['/form', 'edit', item.id]">
                Edit
            </button>
        </td>
    </tr>
</table>
```

```
<button class="btn btn-primary" routerLink="/form/create">
    Create New Product
</button>
<button class="btn btn-danger" (click)="deleteProduct(-1)">
    Generate HTTP Error
</button>
<button class="btn btn-danger" routerLink="/does/not/exist">
    Generate Routing Error
</button>
```

При щелчке на кнопке приложение должно перейти по URL */does/not/exist*, для которого маршрут не настроен. Если соответствие для URL не найдено, генерируется ошибка, которая перехватывается и обрабатывается классом обработки ошибок, что приводит к выдаче предупреждения компонентом сообщений (рис. 26.2).

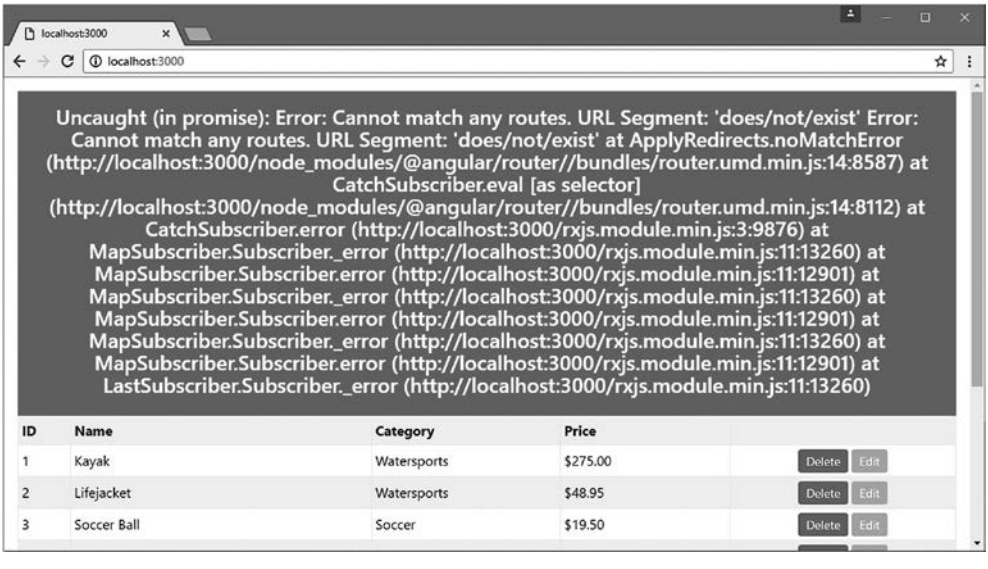


Рис. 26.2. Ошибка навигации по умолчанию

Это не самый полезный вариант обработки неизвестных маршрутов, потому что пользователь не знает, что такое «маршрут», и может не понять, что приложение пытается перейти по некорректному URL-адресу.

Лучше использовать универсальный маршрут для обработки навигации по URL-адресам, которые не были определены, и выбирать компонент, который будет выводить более осмысленное сообщение для пользователя (листинг 26.9).

Листинг 26.9. Включение универсального маршрута в файл app.routing.ts

```
import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "./core/table.component";
import { FormComponent } from "./core/form.component";
import { NotFoundComponent } from "./core/notFound.component";
```

```
const routes: Routes = [
    { path: "form/:mode/:id", component: FormComponent },
    { path: "form/:mode", component: FormComponent },
    { path: "", component: TableComponent },
    { path: "**", component: NotFoundComponent }
]

export const routing = RouterModule.forRoot(routes);
```

Новый универсальный маршрут в листинге используется для выбора компонента `NotFoundComponent`, который выводит сообщение на рис. 26.3 при щелчке на кнопке Generate Routing Error.

Рис. 26.3. Использование универсального маршрута

Кнопка Start Over выполняет навигацию по URL `/`, который выбирает для отображения компонент таблицы.

ПУСТЫЕ И УНИВЕРСАЛЬНЫЕ ПУТИ

Два последних маршрута в листинге 26.9 легко перепутать.

```
...
{ path: "", component: TableComponent },
{ path: "**", component: NotFoundComponent }
...
```

В первом случае используется пустой путь, который совпадает с одним пустым сегментом. Соответствие будет найдено только в том случае, если сегмент пуст. Для нашего маршрута это означает, что соответствие будет найдно только для URL http://localhost:3000/, но позже в этой главе будут приведены другие примеры использования пустого пути для сегментов более сложных URL.

Во втором случае используется универсальный путь, соответствующий любому URL. Порядок маршрутов важен, потому что Angular проверяет их в порядке определения. Универсальный путь всегда должен использоваться в последнем маршруте, потому что последующие маршруты никогда не будут достигнуты.

Эти два пути легко перепутать. Важно запомнить, что совпадение для пустого пути жестко фиксировано (один пустой сегмент), тогда как универсальный путь не столь разборчив и совпадает с любым количеством сегментов независимо от их содержимого.

Перенаправления в маршрутах

Маршруты не обязаны выбирать компоненты; они также могут использоваться как псевдонимы, перенаправляющие браузер на другой URL. Перенаправления определяются при помощи свойства `redirectTo` в маршруте (листинг 26.10).

Листинг 26.10. Перенаправление маршрута в файле app.routing.ts

```
import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "./core/table.component";
import { FormComponent } from "./core/form.component";
import { NotFoundComponent } from "./core/notFound.component";

const routes: Routes = [
    { path: "form/:mode/:id", component: FormComponent },
    { path: "form/:mode", component: FormComponent },
    { path: "does", redirectTo: "/form/create", pathMatch: "prefix" },
    { path: "table", component: TableComponent },
    { path: "", redirectTo: "/table", pathMatch: "full" },
    { path: "**", component: NotFoundComponent }
]

export const routing = RouterModule.forRoot(routes);
```

Свойство `redirectTo` используется для задания URL, на который будет перенаправлен браузер. При определении перенаправлений также должно быть задано свойство `pathMatch` с использованием одного из значений из табл. 26.2.

Таблица 26.2. Значения pathMatch

Имя	Описание
prefix	Значение настраивает маршрут так, чтобы он соответствовал URL-адресам, начинающимся с заданного пути (все последующие сегменты игнорируются)
full	Значение настраивает маршрут так, чтобы он соответствовал только URL-адресам, определяемым свойством path

Первый маршрут, добавленный в листинге 26.10, задает `pathMatch` значение `prefix` с путем `does`; это означает, что он соответствует любому URL-адресу с первым сегментом `does` (например, `/does/not/exist`, как в URL-адресе, по которому производится переход кнопкой Generate Routing Error). Когда браузер переходит по URL-адресу с этим префиксом, система маршрутизации перенаправляет его на URL-адрес `/form/create` (рис. 26.4).

Другие маршруты в листинге 26.10 перенаправляют пустой путь на URL-адрес `/table`, который отображает компонент таблицы. Это стандартный прием, который делает схему URL более очевидной: соответствие находится для URL по умолчанию (*http://localhost:3000/*) с последующим перенаправлением на более содержательный и легче запоминающийся адрес (*http://localhost:3000/table*). В данном случае свойство `pathMatch` содержит значение `full`, хотя это ни на что не влияет, потому что оно применяется к пустому пути.

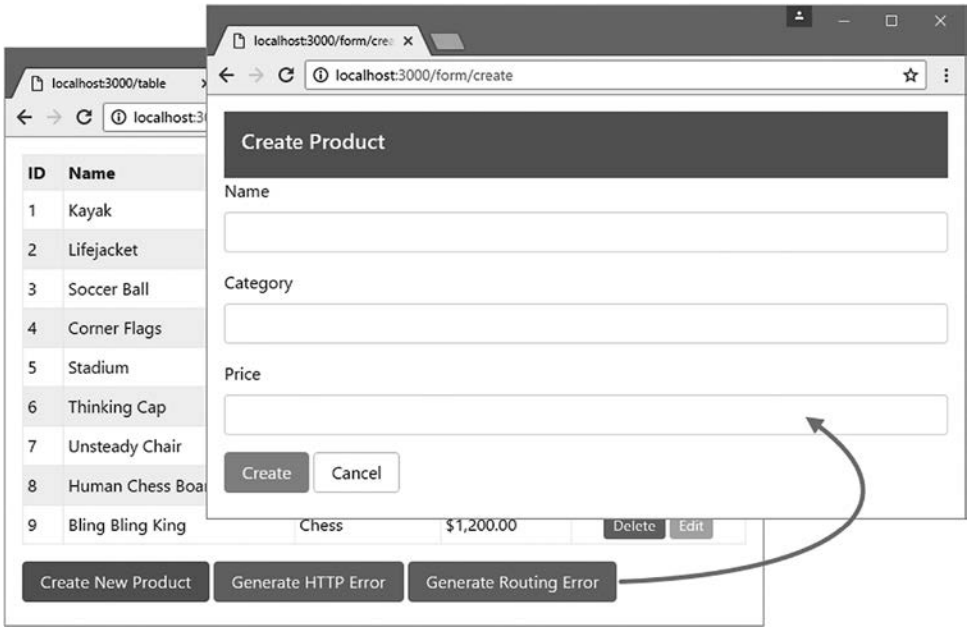


Рис. 26.4. Перенаправление маршрута

Навигация внутри компонента

Во всех примерах навигации в предыдущей главе происходили переходы между разными компонентами: щелчок на кнопке в компоненте таблицы осуществлял переход к компоненту формы, и наоборот.

Это не единственная возможная разновидность навигации: также возможно выполнить навигацию внутри компонента. Для демонстрации листинг 26.11 добавляет к компоненту формы кнопки, позволяющие пользователю отредактировать предыдущий или следующий объект данных.

Листинг 26.11. Добавление кнопок в файле form.component.html

```
<div class="bg-primary p-a-1" [class.bg-warning]="editing">
    <h5>{{editing  ? "Edit" : "Create"}} Product</h5>
</div>
<div *ngIf="editing" class="p-t-1">
    <button class="btn btn-secondary"
        [routerLink]="['/form', 'edit', model.getPreviousProductid(product.id)]">
            Previous
    </button>
    <button class="btn btn-secondary"
        [routerLink]="['/form', 'edit', model.getNextProductId(product.id)]">
            Next
    </button>
</div>
<form novalidate #form="ngForm" (ngSubmit)="submitForm(form)" (reset)="resetForm()" >
```

```
    <div class="form-group">
        <label>Name</label>
        <input class="form-control" name="name"
               [(ngModel)]="product.name" required />
    </div>

    <div class="form-group">
        <label>Category</label>
        <input class="form-control" name="category"
               [(ngModel)]="product.category" required />
    </div>

    <div class="form-group">
        <label>Price</label>
        <input class="form-control" name="price"
               [(ngModel)]="product.price"
               required pattern="^[0-9\.]+$" />
    </div>

    <button type="submit" class="btn btn-primary"
            [class.btn-warning]="editing" [disabled]="form.invalid">
        {{editing ? "Save" : "Create"}}
    </button>
    <button type="reset" class="btn btn-secondary" routerLink="/">Cancel</button>
</form>
```

Эти кнопки содержат привязки для директивы `routerLink` с выражениями, целями которых являются предыдущий и следующий объект модели данных. Таким образом, если, например, щелкнуть на кнопке Edit в таблице для товара `Lifejacket`, то кнопка Next перейдет на URL-адрес для редактирования товара `Soccer Ball`, а кнопка `Previous` — на URL-адрес для товара `Kayak`.

Реакция на текущие изменения маршрутизации

Щелчки на новых кнопках пока ни к чему не приводят. Angular старается действовать эффективно при навигации и знает, что URL-адреса, на которые ведут кнопки Previous и Next, обрабатываются тем же компонентом, который сейчас отображается для пользователя. Вместо того чтобы создавать новый экземпляр компонента, Angular просто сообщает компоненту, что выбранный маршрут изменился.

Проблема возникла из-за того, что компонент формы не настроен для получения уведомлений об изменениях. Его конструктор получает объект `ActivatedRoute`, который используется Angular для передачи подробной информации о текущем маршруте, но при этом используется только его свойство `snapshot`. К тому моменту, когда Angular обновит значения `ActivatedRoute`, конструктор компонента уже давно выполнен; это означает, что уведомление будет пропущено. Такое решение работало, когда конфигурация приложения подразумевала, что новый компонент формы будет создаваться каждый раз, когда пользователь хочет создать или отредактировать товар, но сейчас этого уже недостаточно.

К счастью, класс `ActivatedRoute` определяет набор свойств, при помощи которых заинтересованные стороны могут получать уведомления через объекты Reactive Extensions `Observable`. Эти свойства соответствуют тем, что предоставляются объ-

ектом `ActivatedRouteSnapshot`, возвращаемым свойством `snapshot` (см. главу 25), но отправляют новые события при последующих изменениях (табл. 26.3).

Таблица 26.3. Свойства класса ActivatedRoute, относящиеся к Observable

Имя	Описание
url	Свойство возвращает объект Observable<UrlSegment[]>, который предоставляет набор сегментов URL при каждом изменении маршрута
params	Свойство возвращает объект Observable<Params>, который предоставляет параметры URL при каждом изменении маршрута
queryParams	Свойство возвращает объект Observable<Params>, который предоставляет параметры запроса URL при каждом изменении маршрута
fragment	Свойство возвращает объект Observable<string>, который предоставляет фрагмент URL при каждом изменении маршрута

Свойства могут использоваться компонентами, которым нужно обрабатывать навигационные изменения, не приводящие к отображению другого компонента (листинг 26.12).

ПРИМЕЧАНИЕ

Если вам нужно объединить разные элементы данных из маршрута (например, сегменты и параметры), подпишитесь на объект Observer для одного элемента данных и используйте свойство snapshot для получения остальных необходимых данных.

Листинг 26.12. Отслеживание изменений маршрута в файле form.component.ts

```
import { Component, Inject } from "@angular/core";
import { NgForm } from "@angular/forms";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
import { ActivatedRoute, Router } from "@angular/router";

@Component({
    selector: "paForm",
    moduleId: module.id,
    templateUrl: "form.component.html",
    styleUrls: ["form.component.css"]
})
export class FormComponent {
    product: Product = new Product();

    constructor(private model: Model, activeRoute: ActivatedRoute,
        private router: Router) {

        activeRoute.params.subscribe(params => {
            this.editing = params["mode"] == "edit";
            let id = params["id"];
            if (id != null) {
                Object.assign(this.product, model.getProduct(id) || new Product());
            }
        })
    }
```

```
    editing: boolean = false;

    submitForm(form: NgForm) {
        if (form.valid) {
            this.model.saveProduct(this.product);
            this.router.navigateByUrl("/");
        }
    }

    resetForm() {
        this.product = new Product();
    }
}
```

Компонент подписывается на объект `Observer<Params>`, который отправляет новый объект `Params` подписчикам при каждом изменении маршрута. Объекты `Observer`, возвращаемые свойствами `ActivatedRoute`, отправляют подробную информацию о последнем изменении маршрута при вызове метода `subscribe`; это гарантирует, что конструктор компонента не пропустит исходную навигацию, которая привела к его вызову.

В результате компонент может реагировать на изменения маршрута, которые не заставляют Angular создавать новый компонент, как в случае с кнопками Next и Previous, сменяющими товар, выбранный для редактирования (рис. 26.5).

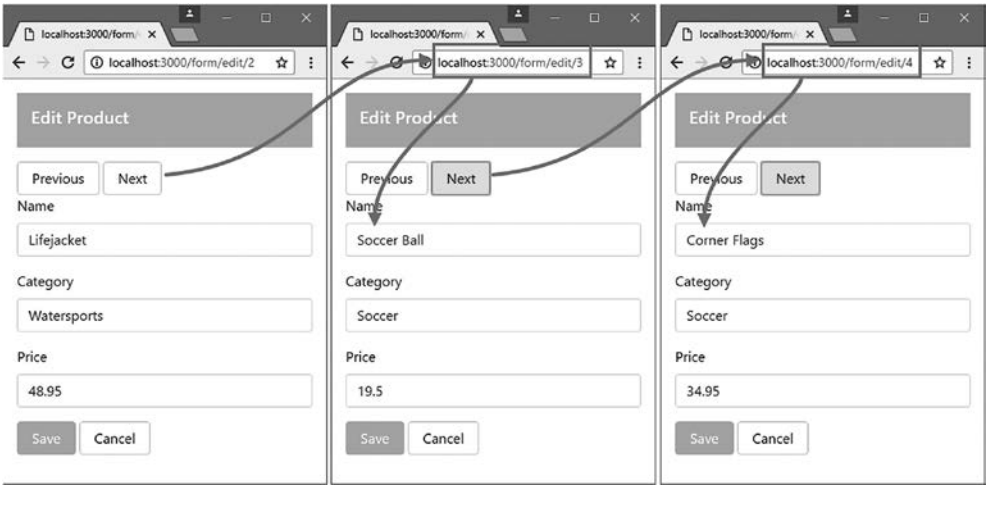


Рис. 26.5. Реакция на изменения маршрута

ПРИМЕЧАНИЕ

Эффект навигации очевиден, когда активизируемый маршрут изменяет отображаемый компонент. Если изменяются только данные, все не столь очевидно. Чтобы выделить изменения, Angular может применить анимации, привлекающие внимание к эффектам навигации. За подробностями обращайтесь к главе 28.

Стилевое оформление ссылок для активных маршрутов

Одно из типичных применений системы маршрутизации — отображение разных навигационных элементов рядом с выбираемым контентом. В листинге 26.12 в приложение добавляется новый маршрут, который позволит выбирать компонент таблицы в URL-адресе, содержащем фильтр категории.

Листинг 26.13. Определение маршрута в файле app.routing.ts

```
import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "./core/table.component";
import { FormComponent } from "./core/form.component";
import { NotFoundComponent } from "./core/notFound.component";

const routes: Routes = [
    { path: "form/:mode/:id", component: FormComponent },
    { path: "form/:mode", component: FormComponent },
    { path: "does", redirectTo: "/form/create", pathMatch: "prefix" },
    { path: "table/:category", component: TableComponent },
    { path: "table", component: TableComponent },
    { path: "", redirectTo: "/table", pathMatch: "full" },
    { path: "**", component: NotFoundComponent }
]

export const routing = RouterModule.forRoot(routes);
```

В листинге 26.13 класс `TableComponent` обновляется так, чтобы он использовал систему маршрутизации для получения информации об активном маршруте и присваивал значение параметра маршрута `category` свойству `category`, к которому можно обращаться в шаблоне. Свойство `category` используется в методе `getProducts` для фильтрации объектов в модели данных.

Листинг 26.14. Добавление поддержки фильтра категории в файле table.component.ts

```
import { Component, Inject } from "@angular/core";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
import { ActivatedRoute } from "@angular/router";

@Component({
    selector: "paTable",
    moduleId: module.id,
    templateUrl: "table.component.html"
})
export class TableComponent {
    category: string = null;

    constructor(private model: Model, activeRoute: ActivatedRoute) {
        activeRoute.params.subscribe(params => {
            this.category = params["category"] || null;
        })
    }
```

```
    getProduct(key: number): Product {
        return this.model.getProduct(key);
    }

    getProducts(): Product[] {
        return this.model.getProducts()
            .filter(p => this.category == null || p.category == this.category);
    }

    get categories(): string[] {
        return this.model.getProducts()
            .map(p => p.category)
            .filter((category, index, array) => array.indexOf(category) == index);
    }

    deleteProduct(key: number) {
        this.model.deleteProduct(key);
    }
}
```

Также появилось новое свойство `categories`, которое будет использоваться в шаблоне для генерирования набора категорий для фильтрации. Остается добавить в шаблон элементы HTML, которые позволят пользователю назначить фильтр (листинг 26.15).

Листинг 26.15. Добавление элементов фильтра в файле table.component.html

```
<div class="col-xs-3">
    <button class="btn btn-secondary btn-block"
            routerLink="/" routerLinkActive="active">
        All
    </button>
    <button *ngFor="let category of categories" class="btn btn-secondary btn-block"
            [routerLink]="['/table', category]" routerLinkActive="active">
        {{category}}
    </button>
</div>
<div class="col-xs-9">
    <table class="table table-sm table-bordered table-striped">
        <tr>
            <th>ID</th><th>Name</th><th>Category</th><th>Price</th><th></th>
        </tr>
        <tr *ngFor="let item of getProducts()">
            <td style="vertical-align:middle">{{item.id}}</td>
            <td style="vertical-align:middle">{{item.name}}</td>
            <td style="vertical-align:middle">{{item.category}}</td>
            <td style="vertical-align:middle">
                {{item.price | currency:"USD":true }}
            </td>
            <td class="text-xs-center">
                <button class="btn btn-danger btn-sm"
                        (click)="deleteProduct(item.id)">
                    Delete
                </button>
                <button class="btn btn-warning btn-sm"
```

```
                    [routerLink]="['/form', 'edit', item.id]">
                    Edit
                </button>
            </td>
        </tr>
    </table>
</div>
<div class="col-xs-12 p-t-1">
    <button class="btn btn-primary" routerLink="/form/create">
        Create New Product
    </button>
    <button class="btn btn-danger" (click)="deleteProduct(-1)">
        Generate HTTP Error
    </button>
    <button class="btn btn-danger" routerLink="/does/not/exist">
        Generate Routing Error
    </button>
</div>
```

Важный аспект этого примера — использование атрибута `routerLinkActive`. Он задает класс CSS, который будет назначен элементу в том случае, если URL-адрес, заданный атрибутом `routerLink`, соответствует текущему маршруту.

В листинге задается класс CSS с именем `active` — класс фреймворка Bootstrap для назначения активного состояния кнопок. В сочетании с функциональностью, добавленной в компонент в листинге 26.14, в результате создается набор кнопок для просмотра товаров одной категории (рис. 26.6).

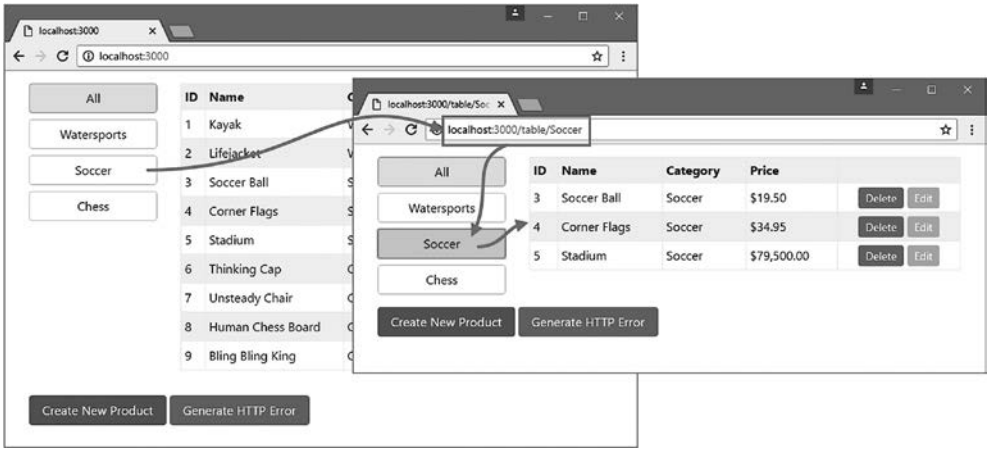


Рис. 26.6. Фильтрация товаров

Если щелкнуть на кнопке Soccer, приложение переходит по URL-адресу `/table/Soccer`, а в таблице выводятся товары только из категории Soccer. Кнопка Soccer также будет выделена, так как атрибут `routerLinkActive` означает, что Angular добавляет элемент `button` в класс Bootstrap `active`.

Исправление всех кнопок

В кнопках навигации проявляется распространенная проблема: кнопке All всегда назначается класс `active`, несмотря на то что пользователь отфильтровал таблицу для вывода конкретной категории.

Это происходит из-за того, что атрибут `routerLinkActive` по умолчанию выполняет частичный поиск совпадения для активных URL-адресов. В нашем примере для URL `/` кнопка All всегда будет активизироваться, потому что эта часть находится в начале всех URL-адресов. Проблему можно решить настройкой директивы `routerLinkActive` (листинг 26.16).

Листинг 26.16. Настройка директивы в файле table.component.html

```
...
<div class="col-xs-3">
    <button class="btn btn-secondary btn-block"
            routerLink="/table" routerLinkActive="active"
            [routerLinkActiveOptions]="{exact: true}">
        All
    </button>
    <button *ngFor="let category of categories" class="btn btn-secondary btn-block"
            [routerLink]="['/table', category]" routerLinkActive="active">
        {{category}}
    </button>
</div>
...
```

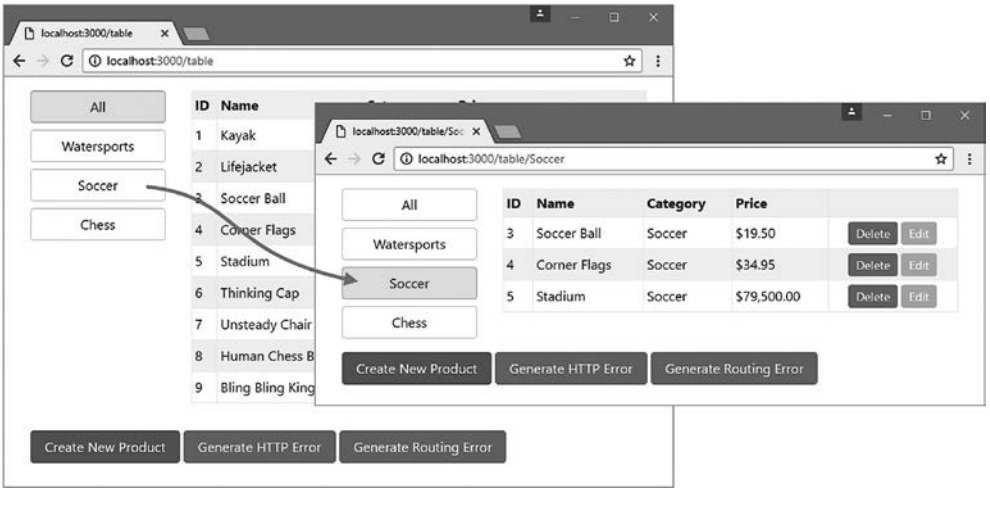


Рис. 26.7. Решение проблемы с кнопкой All

Конфигурация задается с использованием привязки с атрибутом `routerLinkActiveOptions`, который получает объектный литерал. Свойство `exact` — единственно

доступный параметр конфигурации — используется для управления сопоставлением URL-адреса активного маршрута. Если этому свойству задается значение `true`, то элемент включается в класс, заданный атрибутом `routerLinkActive`, только при наличии точного соответствия с URL активного маршрута. С этими изменениями кнопка All будет выделена только при отображении всех товаров (см. рис. 26.7).

Создание дочерних маршрутов

Дочерние маршруты позволяют компонентам реагировать на часть URL-адреса посредством внедрения элементов `router-outlet` в их шаблоны, с созданием более сложной структуры контента. Мы используем простые компоненты, которые были созданы в начале главы, для демонстрации работы дочерних маршрутов. Эти компоненты будут отображаться над таблицей товаров, а отображаемый компонент будет задаваться в URL-адресах из табл. 26.4.

В листинге 26.17 приведены изменения в конфигурации маршрутизации приложения для реализации стратегии маршрутизации, представленной в таблице.

Таблица 26.4. URL-адреса и выбираемые ими компоненты

URL	Компонент
/table/products	Отображается компонент ProductCountComponent
/table/categories	Отображается компонент CategoryCountComponent
/table	Не отображается ни один компонент

Листинг 26.17. Конфигурация маршрутов в файле app.routing.ts

```
import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "./core/table.component";
import { FormComponent } from "./core/form.component";
import { NotFoundComponent } from "./core/notFound.component";
import { ProductCountComponent } from "./core/productCount.component";
import { CategoryCountComponent } from "./core/categoryCount.component";

const routes: Routes = [
    { path: "form/:mode/:id", component: FormComponent },
    { path: "form/:mode", component: FormComponent },
    { path: "does", redirectTo: "/form/create", pathMatch: "prefix" },
    {
        path: "table",
        component: TableComponent,
        children: [
            { path: "products", component: ProductCountComponent },
            { path: "categories", component: CategoryCountComponent },
            { path: "" }
        ]
    },
    { path: "table/:category", component: TableComponent },
    //{ path: "table", component: TableComponent },
```

```
    { path: "", redirectTo: "/table", pathMatch: "full" },
    { path: "**", component: NotFoundComponent }
]

export const routing = RouterModule.forRoot(routes);
```

Дочерние маршруты определяются при помощи свойства `children`, которым задается массив маршрутов, определяемых по аналогии с маршрутами верхнего уровня.

Когда Angular использует весь URL-адрес для сопоставления маршрута с дочерними маршрутами, соответствие будет найдено только в том случае, если URL-адрес, на который переходит браузер, содержит как сегменты, соответствующие сегменту верхнего уровня, так и сегменты, заданные одним из дочерних маршрутов.

Первые два дочерних маршрута используют свойства `path` и `component` для определения маршрутов, которые ищут соответствие сегменту URL и выбирают один из новых компонентов. Третий дочерний маршрут делает нечто иное. Этот маршрут является *бескомпонентным*; это означает, что он использует свойство `path` для поиска соответствия с сегментом URL, но не выбирает компонент для отображения. Это полезный механизм, обеспечивающий более широкое соответствие маршрутов, так как без бескомпонентного маршрута нам пришлось бы создавать отдельный маршрут для URL-адреса `/table`.

ПРИМЕЧАНИЕ

Обратите внимание: новый маршрут добавляется перед маршрутом с путем table/:category. Angular пытается сопоставлять маршруты в порядке их определения. Путь table/:category будет соответствовать URL /table/products и /table/categories, в результате чего компонент таблицы попытается отфильтровать товары по несуществующим категориям. Так как более конкретные маршруты размещаются в начале списка, соответствия для URL /table/products и /table/categories будут найдены до рассмотрения пути table/:category.

Создание элемента router-outlet для дочернего маршрута

Компоненты, выбираемые дочерними маршрутами, отображаются в элементе `router-outlet`, определенном в шаблоне компонента, выбранного родительским маршрутом. В нашем примере это означает, что дочерние маршруты выберут элемент в шаблоне компонента таблицы (листинг 26.18), что также добавляет элементы для навигации по новым маршрутам.

Листинг 26.18. Добавление элемента router-outlet и навигации в файле table.component.html

```
<div class="col-xs-3">
    <button class="btn btn-secondary btn-block"
            routerLink="/" routerLinkActive="active"
            [routerLinkActiveOptions]="{exact: true}">
        All
    </button>
    <button *ngFor="let category of categories" class="btn btn-secondary btn-block"
```

```
            [routerLink]="['/table', category]" routerLinkActive="active">
        {{category}}
    </button>
</div>
<div class="col-xs-9">
    <div class="m-b-1">
        <button class="btn btn-info" routerLink="/table/products">
            Count Products
        </button>
        <button class="btn btn-primary" routerLink="/table/categories">
            Count Categories
        </button>
        <button class="btn btn-secondary" routerLink="/table">
            Count Neither
        </button>
        <div class="m-t-1">
            <router-outlet></router-outlet>
        </div>
    </div>
    <table class="table table-sm table-bordered table-striped">
        <tr>
            <th>ID</th><th>Name</th><th>Category</th><th>Price</th><th></th>
        </tr>
        <tr *ngFor="let item of getProducts()">
            <td style="vertical-align:middle">{{item.id}}</td>
            <td style="vertical-align:middle">{{item.name}}</td>
            <td style="vertical-align:middle">{{item.category}}</td>
            <td style="vertical-align:middle">
                {{item.price | currency:"USD":true }}
            </td>
            <td class="text-xs-center">
                <button class="btn btn-danger btn-sm"
                        (click)="deleteProduct(item.id)">
                    Delete
                </button>
                <button class="btn btn-warning btn-sm"
                    [routerLink]="['/form', 'edit', item.id]">
                    Edit
                </button>
            </td>
        </tr>
    </table>
</div>
<div class="col-xs-12 p-t-1">
    <button class="btn btn-primary" routerLink="/form/create">
        Create New Product
    </button>
    <button class="btn btn-danger" (click)="deleteProduct(-1)">
        Generate HTTP Error
    </button>
    <button class="btn btn-danger" routerLink="/does/not/exist">
        Generate Routing Error
    </button>
</div>
```

Элементы `button` содержат атрибуты `routerLink` с URL-адресами из табл. 26.4. Также имеется элемент `router-outlet`, который будет использоваться для отображения выбранного компонента (рис. 26.8); такого компонента может не быть, если браузер переходит по URL `/table`.

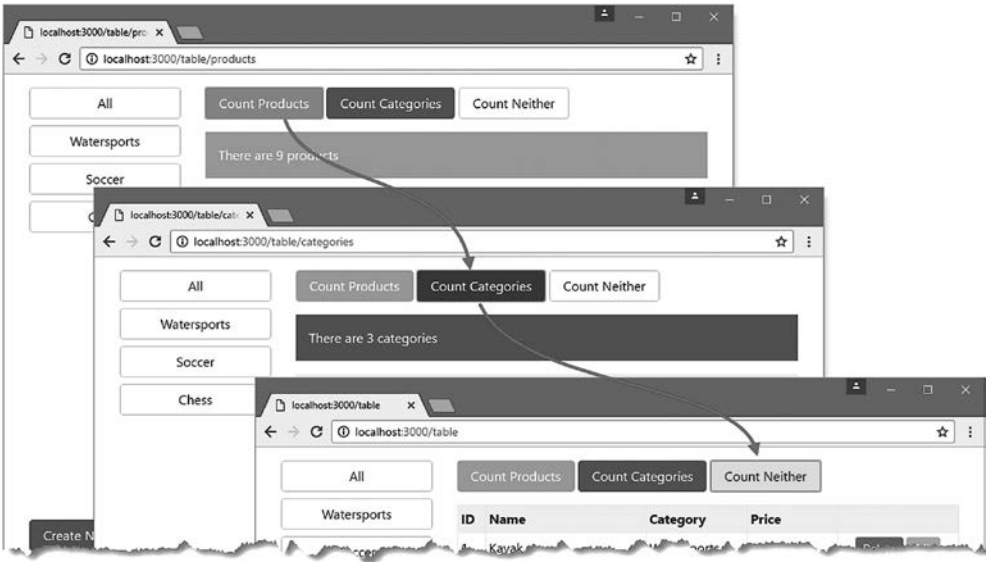


Рис. 26.8. Использование дочерних маршрутов

Обращение к параметрам из дочерних маршрутов

Дочерние маршруты могут использовать всю функциональность, доступную для маршрутов верхнего уровня, включая определение параметров маршрутов и даже наличие собственных дочерних маршрутов. Особого внимания заслуживают параметры маршрутов в дочерних маршрутах — это связано с тем, как Angular отделяет потомков от родителей. В этом разделе мы добавим поддержку URL-адресов из табл. 26.5.

Таблица 26.5. Новые URL-адреса, поддерживаемые в нашем примере

Имя	Описание
/table/:category/products	Маршрут отфильтровывает содержимое таблицы и выбирает ProductCountComponent
/table/:category/categories	Маршрут отфильтровывает содержимое таблицы и выбирает CategoryCountComponent

В листинге 26.19 определяются маршруты, поддерживающие URL-адреса из таблицы.

Листинг 26.19. Название листинга

```
import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "./core/table.component";
import { FormComponent } from "./core/form.component";
import { NotFoundComponent } from "./core/notFound.component";
import { ProductCountComponent } from "./core/productCount.component";
import { CategoryCountComponent } from "./core/categoryCount.component";
const childRoutes: Routes = [
    { path: "products", component: ProductCountComponent },
    { path: "categories", component: CategoryCountComponent },
    { path: "", component: ProductCountComponent }
];
const routes: Routes = [
    { path: "form/:mode/:id", component: FormComponent },
    { path: "form/:mode", component: FormComponent },
    { path: "does", redirectTo: "/form/create", pathMatch: "prefix" },
    { path: "table", component: TableComponent, children: childRoutes },
    { path: "table/:category", component: TableComponent, children: childRoutes },
    { path: "", redirectTo: "/table", pathMatch: "full" },
    { path: "**", component: NotFoundComponent }
]

export const routing = RouterModule.forRoot(routes);
```

Типом свойства `children` является объект `Routes`, что позволяет свести к минимуму дублирование конфигурации маршрута при необходимости применить один набор дочерних маршрутов в разных частях схемы URL. В листинге дочерние маршруты определяются в объекте `Routes` с именем `childRoutes`, который используется как значение свойства `children` в двух разных маршрутах верхнего уровня.

ПРИМЕЧАНИЕ

Обратите внимание: в листинге 26.19 дочерний маршрут пустого пути изменяется для выбора компонента ProductCountComponent. На момент написания книги в Angular существовал дефект, который вызывал ошибку, если система навигации не выбирала компонент. Для простоты я выбираю ProductCountComponent, но если вы не хотите отображать контент для пользователя, создайте компонент с пустым шаблоном и выберите его.

Чтобы сделать возможным выбор этих новых маршрутов, в листинге 26.20 изменяются целевые адреса кнопок, которые отображаются над таблицей, чтобы они выполняли навигацию относительно текущего URL-адреса. Я удалил кнопку Count Neither, так как при совпадении дочернего маршрута пустого пути будет отображаться `ProductCountComponent`.

Листинг 26.20. Использование относительных URL-адресов в файле table.component.html

```
...
<div class="m-b-1">
    <button class="btn btn-info" routerLink="products">Count Products</button>
    <button class="btn btn-primary" routerLink="categories">Count Categories
                                                    </button>
```

```
    <div class="m-t-1">
        <router-outlet></router-outlet>
    </div>
</div>
...
```

Когда Angular занимается поиском соответствия маршрутов, информация, предоставляемая компонентам, выбираемым через объект `ActivatedRoute`, разделяется, так что каждый компонент получает информацию только о той части маршрута, которая выбрала его.

В случае маршрутов, добавленных в листинге 26.20, это означает, что `ProductCountComponent` и `CategoryCountComponent` получают объект `ActivatedRoute`, который описывает только выбравший их дочерний маршрут, с единственным сегментом `/products` или `/categories`. Точно так же компонент `TableComponent` получает объект `ActivatedRoute`, который не содержит сегмент, использовавшийся для подбора дочернего маршрута.

К счастью, класс `ActivatedRoute` предоставляет некоторые свойства, которые дают доступ к остальным частям маршрута, позволяя родителям и потомкам обращаться к остальным данным маршрутизации (табл. 26.6).

Таблица 26.6. Свойства ActivatedRoute для обращения к данным маршрутизации

Имя	Описание
PathFromRoot	Свойство возвращает массив объектов ActivatedRoute, представляющий все маршруты, использованные для сопоставления текущего URL-адреса
parent	Свойство возвращает объект ActivatedRoute, представляющий родителя маршрута, который выбрал компонент
firstChild	Свойство возвращает объект ActivatedRoute, представляющий первый дочерний маршрут, использованный для сопоставления текущего URL-адреса
children	Свойство возвращает массив объектов ActivatedRoute, представляющий все дочерние маршруты, использованные для сопоставления текущего URL-адреса

В листинге 16.21 показано, как компонент `ProductCountComponent` может обратиться к расширенному набору маршрутов, использованных для сопоставления текущего URL-адреса, чтобы получить значение параметра маршрута `category` и адаптировать свой вывод при фильтрации содержимого таблицы для одной категории.

Листинг 26.21. Обращение к данным маршрутов в файле productCount.component.ts

```
import {
    Component, KeyValueDiffer,
    KeyValueDiffers, ChangeDetectorRef
} from "@angular/core";
import { Model } from "../model/repository.model";
import { ActivatedRoute } from "@angular/router";

@Component({
```

```
    selector: "paProductCount",
    template: `<div class="bg-info p-a-1">There are {{count}} products</div>`
})
export class ProductCountComponent {
    private differ: KeyValueDiffer;
    count: number = 0;
    private category: string;

    constructor(private model: Model,
            private keyValueDiffers: KeyValueDiffers,
            private changeDetector: ChangeDetectorRef,
            activeRoute: ActivatedRoute) {

        activeRoute.pathFromRoot.forEach(route => route.params.subscribe(params => {
            if (params["category"] != null) {
                this.category = params["category"];
                this.updateCount();
            }
        }))
    }

    ngOnInit() {
        this.differ = this.keyValueDiffers
            .find(this.model.getProducts())
            .create(this.changeDetector);
    }

    ngDoCheck() {
        if (this.differ.diff(this.model.getProducts()) != null) {
            this.updateCount();
        }
    }

    private updateCount() {
        this.count = this.model.getProducts()
            .filter(p => this.category == null || p.category == this.category)
            .length;
    }
}
```

Свойство `pathFromRoot` особенно полезно, потому что оно позволяет компоненту проанализировать все маршруты, которые были использованы при сопоставлении URL. Angular сводит к минимуму обновления маршрутизации; это означает, что компонент, который был выбран дочерним маршрутом, не получит уведомления об изменениях через свой объект `ActivatedRoute`, если изменился только его родитель. Именно по этой причине я подписался на уведомления от всех объектов `ActivatedRoute`, возвращаемых свойством `pathFromRoot`; это гарантирует, что компонент всегда будет обнаруживать изменения в значении параметра маршрута `category`.

Сохраните изменения и щелкните на кнопке Watersports, чтобы отфильтровать содержимое таблицы. Затем щелкните на кнопке Count Products, которая выбирает

компонент ProductCountComponent. Количество товаров, возвращаемое компонентом, будет соответствовать количеству строк в таблице (рис. 26.9).

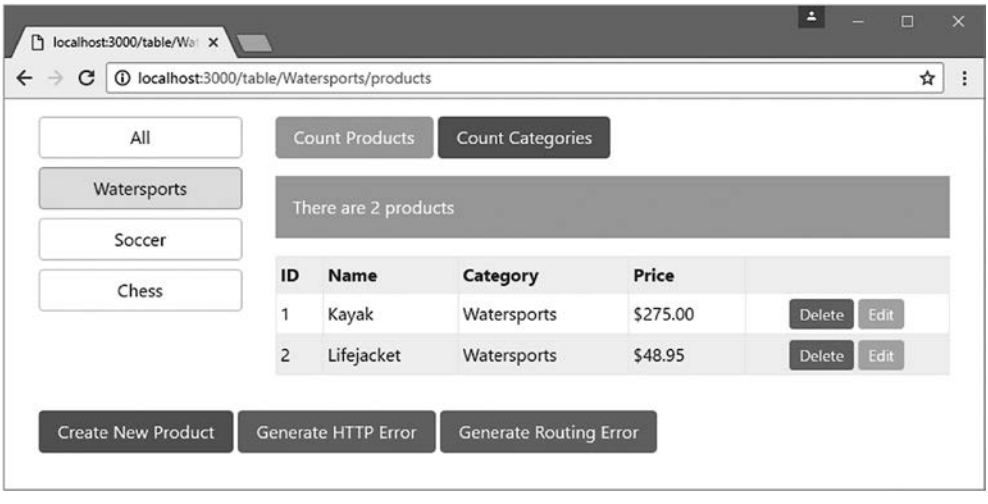


Рис. 26.9. Обращение к другим маршрутам, использованным для поиска соответствия URL

Итоги

В этой главе продолжено описание возможностей системы маршрутизации URL в Angular, выходящих за рамки базовой функциональности предыдущей главы. Вы узнали, как создавать универсальные маршруты и маршруты перенаправления, маршруты для навигации относительно текущего URL-адреса и дочерние маршруты для отображения вложенных компонентов. В следующей главе, завершающей описание системы маршрутизации URL, будут представлены самые нетривиальные возможности.

27 Маршрутизация и навигация: часть 3

В этой главе я продолжу описание системы маршрутизации URL в Angular, уделяя основное внимание самым нетривиальным возможностям. Вы узнаете, как управлять активизацией маршрутов, как динамически загружать функциональные модули и как использовать несколько элементов `router-outlet` в шаблоне. В табл. 27.1 приведена краткая сводка материала главы.

Таблица 27.1. Сводка материала главы

Проблема	Решение	Листинг
Необходимость отложить навигацию до завершения задачи	Используйте резольвер маршрутов	1–7
Предотвращение активизации маршрута	Используйте защитник активизации	8–14
Предотвращение ухода от текущего контента	Используйте защитник деактивизации	15–19
Отложенная загрузка функционального модуля	Создайте динамически загружаемый модуль	20–25
Управление использованием динамически загружаемого модуля	Используйте защитник загрузки	26–28
Использование маршрутизации для управления несколькими элементами `router-outlet`	Используйте именованные элементы `router-outlet` в одном шаблоне	29–34

Подготовка проекта

В этой главе мы продолжим использовать проект exampleApp, созданный в главе 22 и изменявшийся в последующих главах. Для подготовки к задачам этой главы я упростил конфигурацию маршрутизации так, как показано в листинге 27.1.

ПРИМЕЧАНИЕ

Для работы примеров этой главы необходим пакет Reactive Extensions, созданный в главе 22. Если вы не хотите создавать проект самостоятельно, его вместе с модулем Reactive Extensions можно загрузить в составе бесплатно распространяемого архива исходного кода с сайта apress.com.

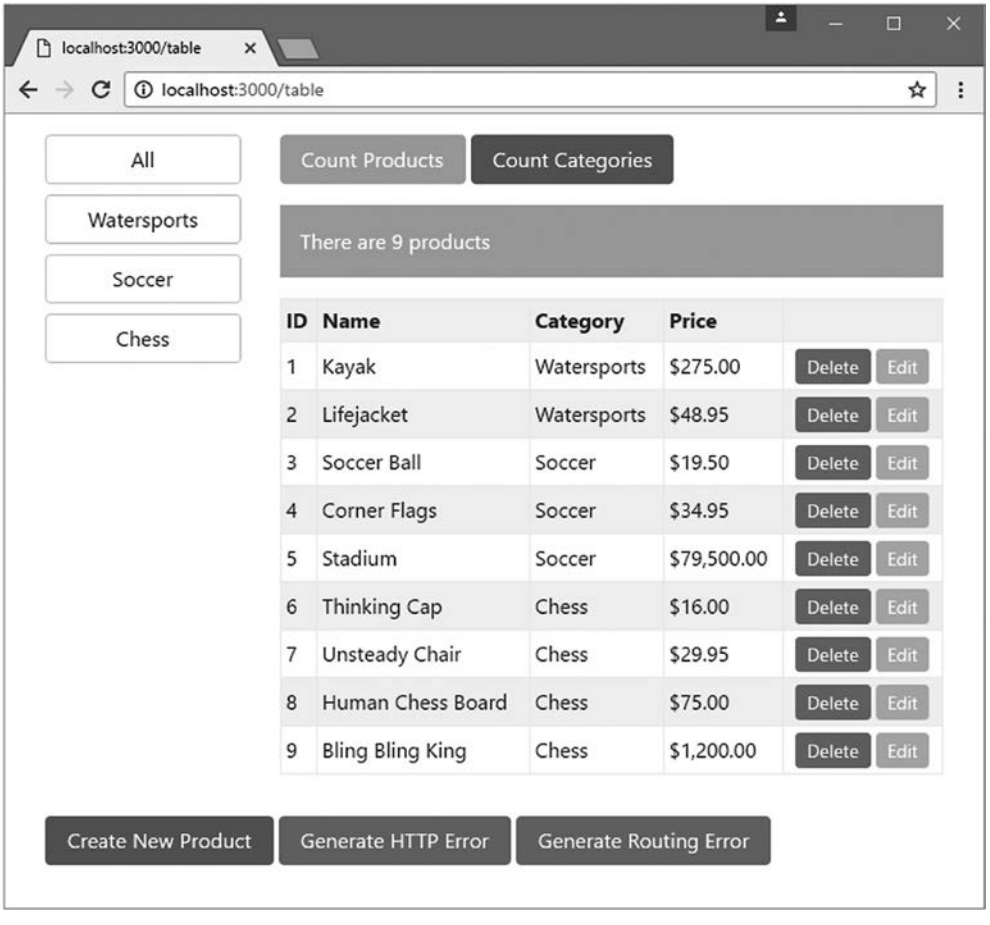


Рис. 27.1. Запуск приложения

Листинг 27.1. Упрощение маршрутов в файле app.routing.ts

```
import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "./core/table.component";
import { FormComponent } from "./core/form.component";
import { NotFoundComponent } from "./core/notFound.component";
import { ProductCountComponent } from "./core/productCount.component";
import { CategoryCountComponent } from "./core/categoryCount.component";

const childRoutes: Routes = [
    { path: "products", component: ProductCountComponent },
    { path: "categories", component: CategoryCountComponent },
    { path: "", component: ProductCountComponent }
];

const routes: Routes = [
```

```
    { path: "form/:mode/:id", component: FormComponent },
    { path: "form/:mode", component: FormComponent },
    { path: "table", component: TableComponent, children: childRoutes },
    { path: "table/:category", component: TableComponent, children: childRoutes },
    { path: "", redirectTo: "/table", pathMatch: "full" },
    { path: "**", component: NotFoundComponent }
]

export const routing = RouterModule.forRoot(routes);
```

Сохраните изменения и выполните следующую команду из папки exampleApp, чтобы запустить компилятор TypeScript, сервер HTTP для разработки и REST-совместимую веб-службу:

```
npm start
```

Открывается новое окно (или вкладка) браузера с контентом, показанным на рис. 27.1.

Защитники маршрутов

На данный момент пользователь может перейти по любому адресу приложения. Это не всегда хорошая идея: либо потому, что приложение может быть не готово, либо потому, что доступ к некоторым частям приложения может быть перекрыт до выполнения конкретных действий. Для управления использованием навигации в Angular поддерживаются *защитники* (guards), которые задаются как часть конфигурации маршрутов в свойствах, определяемых классом `Router` (табл. 27.2).

Таблица 27.2. Свойства маршрутов для использования защитников

Имя	Описание
resolve	Свойство используется для назначения защитников, откладывающих активизацию маршрута до завершения некоторой операции (например, загрузки данных с сервера)
canActivate	Свойство используется для назначения защитников, которые будут использоваться для проверки возможности активизации маршрута
canActivateChild	Свойство используется для назначения защитников, которые будут использоваться для проверки возможности активизации дочернего маршрута
canDeactivate	Свойство используется для назначения защитников, которые будут использоваться для проверки возможности деактивизации маршрута
canLoad	Свойство используется для защиты маршрутов, динамически загружающих функциональные модули (см. раздел «Динамическая загрузка функциональных модулей»)

Отложенная навигация с использованием резольвера

Чаще всего защитники маршрутов применяются для того, чтобы приложение гарантированно получило необходимые данные перед активизацией маршрута. Наш пример асинхронно получает данные от REST-совместимой веб-службы; это означает, что между моментом, когда браузер отправляет запрос HTTP, и моментом получения ответа и обработкой данных может существовать задержка. Возможно, вы не обратили внимания на эту задержку в ходе работы над примерами, потому что браузер и веб-служба работают на одной машине. В реально эксплуатируемом приложении вероятность такой задержки, обусловленной перегрузкой сети, высокой нагрузкой на сервер и десятками других факторов, существенно выше.

Для имитации перегрузки сети в листинге 27.2 класс REST-совместимого источника данных изменен в целях введения задержки после получения ответа от веб-службы.

Листинг 27.2. Введение задержки в файле rest.datasource.ts

```
import { Injectable, Inject, OpaqueToken } from "@angular/core";
import { Http, Request, RequestMethod, Headers, Response } from "@angular/http";
import { Observable } from "rxjs/Observable";
import { Product } from "./product.model";
import "rxjs/add/operator/map";
import "rxjs/add/operator/catch";
import "rxjs/add/observable/throw";
import "rxjs/add/operator/delay";

export const REST_URL = new OpaqueToken("rest_url");

@Injectable()
export class RestDataSource {

    constructor(private http: Http,
        @Inject(REST_URL) private url: string) { }

    // ...другие методы опущены для краткости...

    private sendRequest(verb: RequestMethod,
        url: string, body?: Product): Observable<Product> {
        let headers = new Headers();
        headers.set("Access-Key", "<secret>");
        headers.set("Application-Names", ["exampleApp", "proAngular"]);

        return this.http.request(new Request({
            method: verb,
            url: url,
            body: body,
            headers: headers
        })).delay(5000)
           .map(response => response.json())
           .catch((error: Response) => Observable.throw(
               `Network Error: ${error.statusText} (${error.status})`));
    }
}
```

Вызов метода Reactive Extensions `delay` создает 5-секундную задержку — достаточную для создания заметной паузы, но не слишком продолжительную, чтобы не создавать неудобств при перезагрузке приложения. Чтобы изменить задержку, увеличьте или уменьшите аргумент метода `delay` (значение которого задается в миллисекундах).

В результате задержки пользователь получает неполный и нелогичный макет в то время, когда приложение ожидает загрузки данных (рис. 27.2).

ПРИМЕЧАНИЕ

Задержка применяется ко всем запросам HTTP. Это означает, что при создании, редактировании или удаления товара внесенные изменения не будут отражаться в таблице товаров еще 5 секунд.

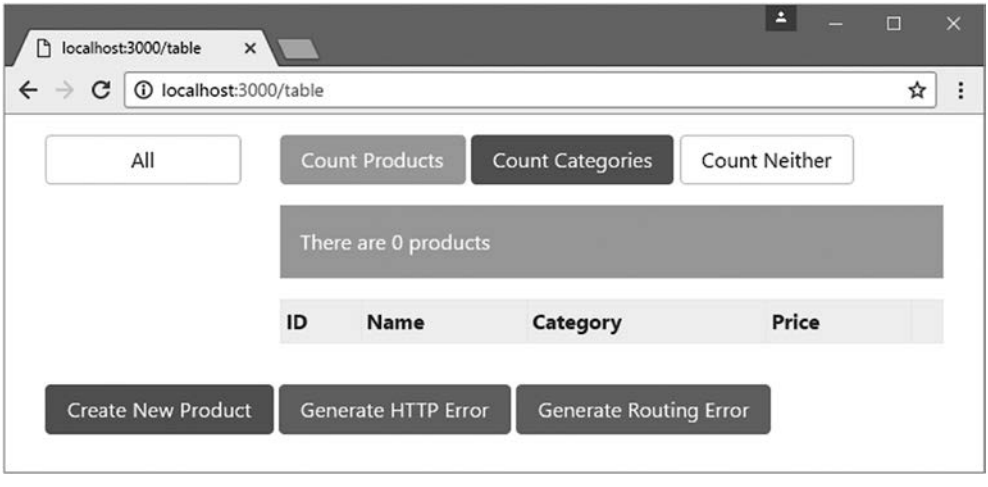


Рис. 27.2. Ожидание данных

Создание службы резольвера

Резольвер гарантирует, что задача будет выполнена перед активизацией маршрута. Чтобы создать резольвер, создайте файл model.resolver.ts в папке exampleApp/app/model и включите в него определение класса из листинга 27.3.

Листинг 27.3. Содержимое файла model.resolver.ts в папке exampleApp/app/model

```
import { Injectable } from "@angular/core";
import { ActivatedRouteSnapshot, RouterStateSnapshot } from "@angular/router";
import { Observable } from "rxjs/Observable";
import { Model } from "./repository.model"
import { RestDataSource } from "./rest.datasource";
import { Product } from "./product.model";

@Injectable()
```

```
export class ModelResolver {

    constructor(
        private model: Model,
        private dataSource: RestDataSource) { }

    resolve(route: ActivatedRouteSnapshot,
            state: RouterStateSnapshot): Observable<Product[]> {

        return this.model.getProducts().length == 0
            ? this.dataSource.getData() : null;
    }
}
```

Резольверы — классы, определяющие метод `resolve` с двумя аргументами. Первый аргумент содержит объект `ActivatedRouteSnapshot` с описанием маршрута, по которому происходит переход, с использованием свойств из главы 25. Второй аргумент содержит объект `RouterStateSnapshot`, который описывает текущий маршрут при помощи одного свойства `url`. Эти аргументы могут использоваться для адаптации резольвера к готовящейся операции навигации, хотя ни один из них не нужен резольверу из листинга; этот резольвер использует одно поведение независимо от маршрутов, на которые (или с которых) выполняется переход.

ПРИМЕЧАНИЕ

Все защитники, описанные в этой главе, могут реализовать интерфейс, описанный в модуле @angular/router. Например, резольверы могут реализовать интерфейс с именем Resolve. Эти интерфейсы необязательны, и в этой главе они не используются.

Метод `resolve` может возвращать результаты трех разных типов (табл. 27.3).

Таблица 27.3. Допустимые типы результатов метода resolve

Тип результата	Описание
Observable<any>	Браузер активизирует новый маршрут при выдаче события объектом Observer
Promise<any>	Браузер активизирует новый маршрут при обработке Promise
Любой другой результат	Браузер активизирует новый маршрут сразу же после того, как метод вернет результат

Результаты `Observable` и `Promise` полезны при использовании асинхронных операций, например запроса данных с использованием запросов HTTP. Angular ожидает завершения асинхронной операции перед активизацией нового маршрута. Любой другой результат интерпретируется как результат синхронной операции, и Angular немедленно активизирует новый маршрут.

Резольвер из листинга 27.3 использует свой конструктор для получения объектов `Model` и `RestDataSource` через механизм внедрения зависимостей. При вызове метод `resolve` проверяет количество объектов в модели данных, чтобы определить,

был ли завершен запрос HTTP к REST-совместимой веб-службе. Если в модели данных нет объектов, то метод `resolve` возвращает объект `Observable` от метода `RestDataSource.getData`, который выдает событие при завершении запроса HTTP. Angular подписывается на Observable и откладывает активизацию нового маршрута до выдачи события. Метод `resolve` возвращает `null`, если в модели есть объекты, и поскольку это ни `Observable`, ни `Promise`, Angular активизирует новый маршрут немедленно.

ПРИМЕЧАНИЕ

Объединение асинхронных и синхронных результатов означает, что резольвер будет откладывать навигацию только до завершения запроса HTTP и заполнения модели данных. Это важно, потому что метод resolve будет вызываться каждый раз, когда приложение попытается выполнить навигацию по маршруту, к которому был применен резольвер.

Регистрация службы резольвера

Затем необходимо зарегистрировать резольвер как службу в функциональном модуле (листинг 27.4).

Листинг 27.4. Регистрация резольвера как службы в файле model.module.ts

```
import { NgModule } from "@angular/core";
import { HttpModule, JsonpModule } from "@angular/http"
import { Model } from "./repository.model";
import { RestDataSource, REST_URL } from "./rest.datasource";
import { ModelResolver } from "./model.resolver";

@NgModule({
    imports: [HttpModule, JsonpModule],
    providers: [Model, RestDataSource, ModelResolver,
        { provide: REST_URL, useValue: "http://localhost:3500/products" }]

})
export class ModelModule { }
```

Применение резольвера

Для применения резольвера к маршрутам используется свойство `resolve` (листинг 27.5).

Листинг 27.5. Применение резольвера в файле app.routing.ts

```
import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "./core/table.component";
import { FormComponent } from "./core/form.component";
import { NotFoundComponent } from "./core/notFound.component";
import { ProductCountComponent } from "./core/productCount.component";
import { CategoryCountComponent } from "./core/categoryCount.component";
import { ModelResolver } from "./model/model.resolver";

const childRoutes: Routes = [
```

```
    {   path: "",
        children: [{ path: "products", component: ProductCountComponent },
                   { path: "categories", component: CategoryCountComponent },
                   { path: "", component: ProductCountComponent }],
        resolve: { model: ModelResolver }
    }
];
```

```
const routes: Routes = [
    { path: "form/:mode/:id", component: FormComponent },
    { path: "form/:mode", component: FormComponent },
    { path: "table", component: TableComponent, children: childRoutes },
    { path: "table/:category", component: TableComponent, children: childRoutes },
    { path: "", redirectTo: "/table", pathMatch: "full" },
    { path: "**", component: NotFoundComponent }
]

export const routing = RouterModule.forRoot(routes);
```

Свойство `resolve` получает объект `map`, значениями свойств которого являются классы-резольверы, применяемые к маршруту. (Имена свойств роли не играют.) Я хочу применить резольвер ко всем представлениям, отображающим таблицу товаров; чтобы избежать дублирования, я создал бескомпонентный путь со свойством `resolve` и использовал его как родителя для существующих дочерних маршрутов.

Отображение временного контента

Angular использует резольвер перед активизацией любого из маршрутов, к которым он был применен; это делается для того, чтобы пользователь не видел таблицу товаров пока модель не будет заполнена данными от REST-совместимой веб-службы. К сожалению, это также означает, что пока браузер ожидает ответа сервера, пользователь будет видеть пустое окно. Для решения этой проблемы в листинге 27.6 служба сообщений уведомляет пользователя о том, что происходит во время загрузки данных.

Листинг 27.6. Вывод сообщения в файле model.resolver.ts

```
import { Injectable } from "@angular/core";
import { ActivatedRouteSnapshot, RouterStateSnapshot } from "@angular/router";
import { Observable } from "rxjs/Observable";
import { Model } from "./repository.model"
import { RestDataSource } from "./rest.datasource";
import { Product } from "./product.model";
import { MessageService } from "../messages/message.service";
import { Message } from "../messages/message.model";

@Injectable()
export class ModelResolver {

    constructor(
        private model: Model,
```

```
        private dataSource: RestDataSource,
        private messages: MessageService) { }

    resolve(route: ActivatedRouteSnapshot,
        state: RouterStateSnapshot): Observable<Product[]> {

        if (this.model.getProducts().length == 0) {
            this.messages.reportMessage(new Message("Loading data..."));
            return this.dataSource.getData();
        }
    }
}
```

Компонент, который выводит сообщение от службы, очищает свое содержимое при получении события `NavigationEnd`. Это означает, что временный контент будет удален после загрузки данных (рис. 27.3).

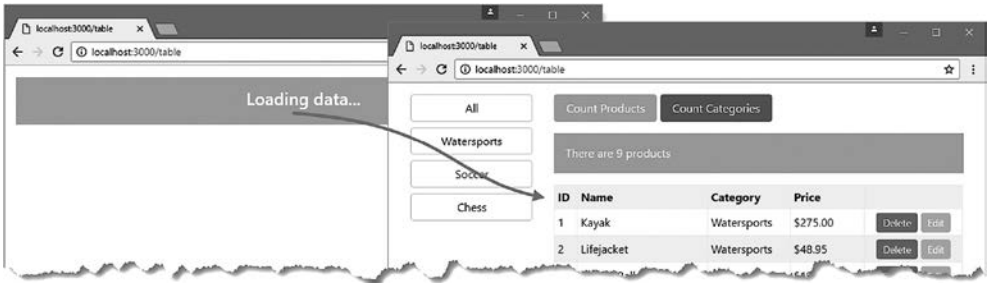


Рис. 27.3. Использование резольвера для отслеживания загрузки данных

Использование резольвера для предотвращения проблем со вводом URL

Как объяснялось в главе 25, при получении запроса URL, для которого нет соответствующего файла, сервер HTTP для разработки возвращает содержимое файла index.html. В сочетании с функциональностью автоматической перезагрузки браузера можно легко внести изменение в проект и заставить браузер перезагрузить URL, который заставляет приложение перейти к конкретному URL-адресу без прохождения необходимых этапов навигации, в ходе которых настраиваются данные состояния.

Щелкните на одной из кнопок Edit в таблице товаров, а затем перезагрузите страницу браузера. Браузер запрашивает URL-адрес (например, *http://localhost:3000/form/edit/1*), но это не приводит к желаемому эффекту, потому что компонент активного маршрута пытается прочитать объект из модели до того, как будет получен ответ HTTP от REST-совместимого сервера. Из-за этого форма остается пустой (рис. 27.4).

Для предотвращения этой проблемы резольвер можно применить более широко, чтобы он защищал другие маршруты (листинг 27.7).

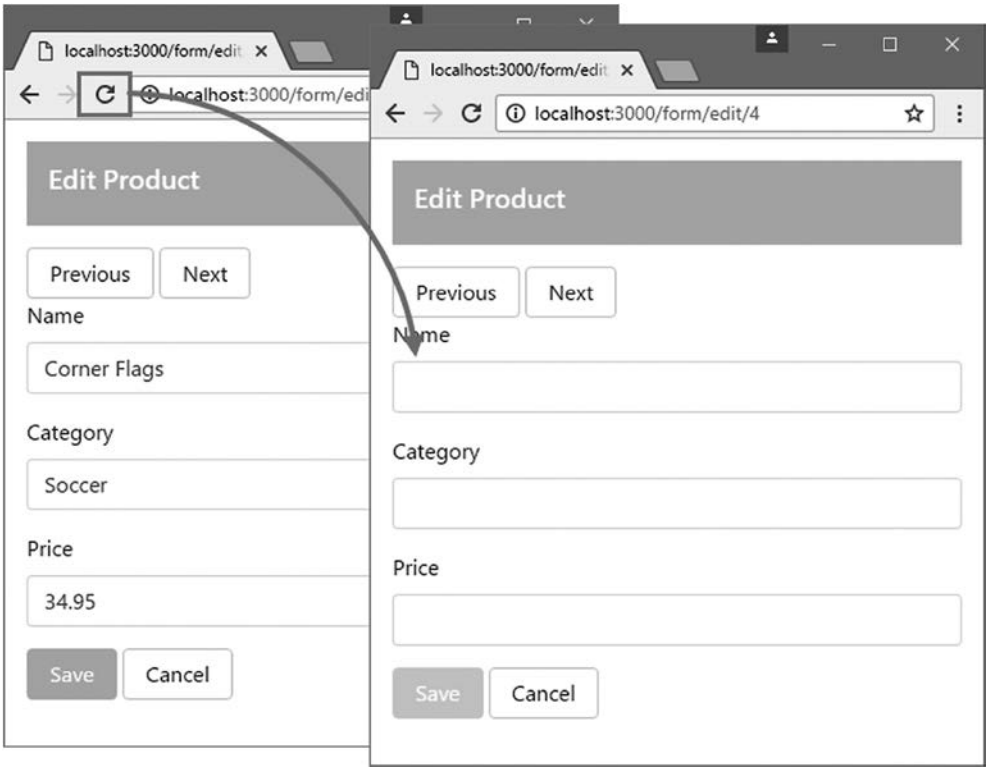


Рис. 27.4. Эффект перезагрузки произвольного URL-адреса

Листинг 27.7. Применение резольвера к другим маршрутам в файле app.routing.ts

```
import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "./core/table.component";
import { FormComponent } from "./core/form.component";
import { NotFoundComponent } from "./core/notFound.component";
import { ProductCountComponent } from "./core/productCount.component";
import { CategoryCountComponent } from "./core/categoryCount.component";
import { ModelResolver } from "./model/model.resolver";

const childRoutes: Routes = [
    {
        path: "",
        children: [{ path: "products", component: ProductCountComponent },
                   { path: "categories", component: CategoryCountComponent },
                   { path: "", component: ProductCountComponent }],
        resolve: { model: ModelResolver }
    }
];

const routes: Routes = [
    {
        path: "form/:mode/:id", component: FormComponent,
```

```
        resolve: { model: ModelResolver }
    },
    {
        path: "form/:mode", component: FormComponent,
        resolve: { model: ModelResolver }
    },
    { path: "table", component: TableComponent, children: childRoutes },
    { path: "table/:category", component: TableComponent, children: childRoutes },
    { path: "", redirectTo: "/table", pathMatch: "full" },
    { path: "**", component: NotFoundComponent }
]

export const routing = RouterModule.forRoot(routes);
```

Применение класса `ModelResolver` к маршрутам, целью которых является `FormComponent`, предотвращает проблему, показанную на рис. 27.4. Существуют и другие решения, включая прием, использованный в приложении SportsStore из главы 8; в нем используется функциональность защитников маршрутов, описанная в разделе «Предотвращение активизации маршрута» этой главы.

Блокировка навигации с использованием защитников

Резольверы используются для задержки навигации на то время, пока приложение выполняет некую подготовительную работу (например, загрузку данных). Другие защитники, предоставляемые Angular, используются для управления самой возможностью выполнения навигации, что может быть полезно, если вы хотите предложить пользователю отменить потенциально нежелательную операцию (например, приводящую к потере изменений в данных) или же ограничить доступ к отдельным частям приложения, если приложение не находится в некотором состоянии (скажем, если пользователь прошел аутентификацию).

Часто при использовании защитников маршрутов требуются дополнительные взаимодействия с пользователем — либо для получения явного разрешения на выполнение операции, либо для получения дополнительных данных (например, удостоверений аутентификации). В этой главе взаимодействия такого рода будут обрабатываться расширением службы сообщений, чтобы сообщения могли запрашивать пользовательский ввод. В листинге 27.8 в класс модели добавляется необязательный аргумент конструктора — свойство `responses`, что позволяет включать в сообщения подсказки для пользователя и обратные вызовы. Свойство `responses` представляет собой массив кортежей (tuples) TypeScript: первое значение содержит имя ответа, который будет выведен для пользователя, а второе — функцию обратного вызова, в аргументе которой будет передаваться имя.

Листинг 27.8. Добавление ответов в файле message.model.ts

```
export class Message {

    constructor(private text: string,
        private error: boolean = false,
        private responses?: [[string, (string) => void]]) { }
}
```

Для реализации этой функции необходимо внести еще только одно изменение: нужно вывести варианты ответов для пользователя. В листинге 27.9 элементы `button` добавляются под текстом сообщения для каждого ответа. Щелчки на кнопках активизируют функцию обратного вызова.

Листинг 27.9. Представление ответов в файле message.component.html

```
<div *ngIf="lastMessage"
     class="bg-info p-a-1 text-xs-center"
     [class.bg-danger]="lastMessage.error">
        <h4>{{lastMessage.text}}</h4>
</div>
<div class="text-xs-center m-b-1">
    <button *ngFor="let resp of lastMessage?.responses; let i = index"
            (click)="resp[1](resp[0])"
            class="btn btn-primary m-a-1" [class.btn-secondary]="i > 0">
        {{resp[0]}}
    </button>
</div>
```

Предотвращение активизации маршрута

Защитники используются для предотвращения активизации маршрута, помогая предотвратить переход приложения в нежелательное состояние или предупредить пользователя о последствиях выполнения операции. Для демонстрации мы защитим URL /form/create, чтобы нельзя было запустить процесс создания нового товара, пока пользователь не согласится на условия обслуживания.

Защитники активизации маршрутов — классы, определяющие метод `canActivate`, который получает те же аргументы `ActivatedRouteSnapshot` и `RouterStateSnapshot`, что и резольверы. Метод `canActivate` может быть реализован для возвращения трех разных типов (табл. 27.4).

Таблица 27.4. Типы результатов, поддерживаемых методом canActivate

Тип результата	Описание
boolean	Этот тип результата обычно используется при синхронной проверке возможности активизации маршрута. С результатом true маршрут активизируется, а с результатом false — нет, что приводит к фактическому игнорированию навигационного запроса
Observable<boolean>	Этот тип результата обычно используется при асинхронной проверке возможности активизации маршрута. Angular ожидает, пока Observable выдаст значение, которое будет использоваться для проверки возможности активизации маршрута. При использовании этого типа результата важно завершить ожидание Observable вызовом метода complete; в противном случае Angular продолжит ожидание
Promise<boolean>	Этот тип результата обычно используется при асинхронной проверке возможности активизации маршрута. Angular ожидает, пока объект Promise будет обработан, и активизирует маршрут при получении значения true. Если же будет получен результат false, это означает, что маршрут не будет активизирован, что приводит к фактическому игнорированию навигационного запроса

Для начала создайте файл terms.guard.ts в папке exampleApp/app и включите в него определение класса из листинга 27.10.

Листинг 27.10. Содержимое файла terms.guard.ts в папке exampleApp/app

```
import { Injectable } from "@angular/core";
import {
    ActivatedRouteSnapshot, RouterStateSnapshot,
    Router
} from "@angular/router";
import { MessageService } from "./messages/message.service";
import { Message } from "./messages/message.model";

@Injectable()
export class TermsGuard {

    constructor(private messages: MessageService,
                private router: Router) { }

    canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot):
        Promise<boolean> | boolean {

        if (route.params["mode"] == "create") {

            return new Promise<boolean>((resolve, reject) => {
                let responses: [[string, (string) => void]] = [
                    ["Yes", () => { resolve(true) }],
                    ["No", () => {
                        this.router.navigateByUrl(this.router.url);
                        resolve(false);
                    }]
                ];
                this.messages.reportMessage(
                    new Message("Do you accept the terms & conditions?",
                        false, responses));
            });
        } else {
            return true;
        }
    }
}
```

Метод `canActivate` может вернуть два разных типа результатов. Первый тип, `boolean`, позволяет защитнику немедленно реагировать на маршруты, которые не нужно защищать; в данном случае это все маршруты без параметра с именем `mode`, значение которого равно `create`. Если URL-адрес, соответствующий маршруту, не содержит этот параметр, то метод `canActivate` возвращает `true`, тем самым приказывая Angular активизировать маршрут. Это важно, потому что функции редактирования и создания зависят от одних маршрутов и защитник не должен мешать операциям редактирования.

Другой тип результата — `Promise<boolean>` — был использован вместо `Observable<true>` для разнообразия. Объект `Promise` использует изменения в службе сообщений для получения ответа от пользователя, подтверждающего, что пользователь согласен на (еще не заданные) условия. Есть два возможных варианта

ответа от пользователя. Если пользователь щелкает на кнопке Yes, то `Promise` разрешается со значением `true`; это приказывает Angular активизировать маршрут и отобразить форму, используемую для создания нового товара. Если же пользователь щелкает на кнопке No, объект `Promise` разрешается со значением `false`, тем самым приказывая Angular игнорировать навигационный запрос.

ОБХОДНОЕ РЕШЕНИЕ

В листинге 27.10 есть одна команда, о которой стоит упомянуть особо. Эта команда выделена в следующем блоке кода, который выполняется при выборе пользователем ответа No:

```
...
["No", () => {
    this.router.navigateByUrl(this.router.url);
    resolve(false);
}]
...
```

При отмене активизации маршрута Angular не разрешает повторно активизировать маршрут. Это изменение появилось в последней версии, выпущенной во время написания книги. Таким образом, после того как пользователь выберет ответ No, повторный щелчок на кнопке Create New Product ни к чему не приведет. Чтобы обойти эту проблему, выделенная команда осуществляет переход по текущему URL-адресу с единственным результатом: отмененную ранее активизацию маршрута теперь можно повторить снова. Возможно, это поведение изменится к тому моменту, когда вы будете читать эту главу. В этом случае вы можете убрать эту команду из своего проекта.

В листинге 27.11 `TermsGuard` регистрируется как служба для использования в конфигурации маршрутизации приложения.

Листинг 27.11. Регистрация защитника как службы в файле app.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { ModelModule } from "./model/model.module";
import { CoreModule } from "./core/core.module";
import { TableComponent } from "./core/table.component";
import { FormComponent } from "./core/form.component";
import { MessageModule } from "./messages/message.module";
import { MessageComponent } from "./messages/message.component";
import { routing } from "./app.routing";
import { AppComponent } from "./app.component";
import { TermsGuard } from "./terms.guard"

@NgModule({
    imports: [BrowserModule, CoreModule, MessageModule, routing],
    declarations: [AppComponent],
    providers: [TermsGuard],
    bootstrap: [AppComponent]
})
export class AppModule { }
```

Наконец, листинг 27.12 применяет защитника к конфигурации маршрутизации. Защитники активизации применяются к маршруту при помощи свойства `canActivate`, которому присваивается массив служб-защитников. Перед тем как Angular активизирует маршрут, метод `canActivate` всех защитников должен вернуть `true` (или `Observable`, или объект `Promise`, при разрешении которого дается результат `true`).

Листинг 27.12. Применение защитника к маршруту в файле app.routing.ts

```
import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "./core/table.component";
import { FormComponent } from "./core/form.component";
import { NotFoundComponent } from "./core/notFound.component";
import { ProductCountComponent } from "./core/productCount.component";
import { CategoryCountComponent } from "./core/categoryCount.component";
import { ModelResolver } from "./model/model.resolver";
import { TermsGuard } from "./terms.guard";

const childRoutes: Routes = [
    {
        path: "",
        children: [{ path: "products", component: ProductCountComponent },
                   { path: "categories", component: CategoryCountComponent },
                   { path: "", component: ProductCountComponent }],
        resolve: { model: ModelResolver }
    }
];

const routes: Routes = [
    {
        path: "form/:mode/:id", component: FormComponent,
        resolve: { model: ModelResolver }
    },
    {
        path: "form/:mode", component: FormComponent,
        resolve: { model: ModelResolver },
        canActivate: [TermsGuard]
    },
    { path: "table", component: TableComponent, children: childRoutes },
    { path: "table/:category", component: TableComponent, children: childRoutes },
    { path: "", redirectTo: "/table", pathMatch: "full" },
    { path: "**", component: NotFoundComponent }
]

export const routing = RouterModule.forRoot(routes);
```

В результате создания и применения защитника активизации пользователю предлагается подтвердить нажатие кнопки Create New Product (рис. 27.5). Если он щелкнет на кнопке Yes, выполняется навигационный запрос и Angular активизирует маршрут для выбора компонента формы, что позволяет создать новый продукт. Если пользователь щелкает на кнопке No, запрос на навигацию будет отменен. В обоих случаях система маршрутизации генерирует событие, которое будет получено компонентом, выводящим сообщение для пользователя. Компонент стирает изображение, чтобы пользователь не видел устаревшую информацию.

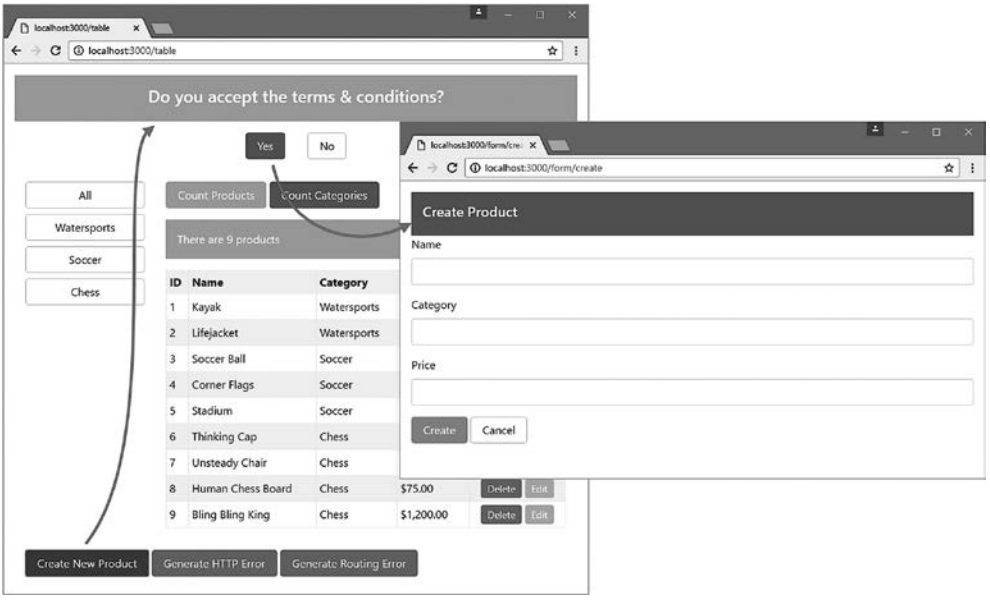


Рис. 27.5. Защита активизации маршрута

Консолидация защитников дочерних маршрутов

Если у вас имеется набор дочерних маршрутов, для защиты от их активизации можно воспользоваться защитником дочерних маршрутов — классом, определяющим метод с именем `canActivateChild`. Защитник применяется к родительскому маршруту в конфигурации приложения, а метод `canActivateChild` вызывается каждый раз, когда должен активизироваться любой из дочерних маршрутов. Метод получает те же объекты `ActivatedRouteSnapshot` и `RouterStateSnapshot`, что и другие защитники, и может возвращать набор типов результатов, перечисленных в табл. 27.4.

Защита маршрута в данном случае проще реализуется изменением конфигурации перед реализацией метода `canActivateChild` (листинг 27.13).

Листинг 27.13. Защита дочерних маршрутов в файле app.routing.ts

```
import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "./core/table.component";
import { FormComponent } from "./core/form.component";
import { NotFoundComponent } from "./core/notFound.component";
import { ProductCountComponent } from "./core/productCount.component";
import { CategoryCountComponent } from "./core/categoryCount.component";
import { ModelResolver } from "./model/model.resolver";
import { TermsGuard } from "./terms.guard";

const childRoutes: Routes = [
    {
```

```
        path: "",
        canActivateChild: [TermsGuard],
        children: [{ path: "products", component: ProductCountComponent },
                   { path: "categories", component: CategoryCountComponent },
                   { path: "", component: ProductCountComponent }],
        resolve: { model: ModelResolver }
    }
];

const routes: Routes = [
    {
        path: "form/:mode/:id", component: FormComponent,
        resolve: { model: ModelResolver }
    },
    {
        path: "form/:mode", component: FormComponent,
        resolve: { model: ModelResolver },
        canActivate: [TermsGuard]
    },
    { path: "table", component: TableComponent, children: childRoutes },
    { path: "table/:category", component: TableComponent, children: childRoutes },
    { path: "", redirectTo: "/table", pathMatch: "full" },
    { path: "**", component: NotFoundComponent }
]

export const routing = RouterModule.forRoot(routes);
```

Защитники дочерних маршрутов применяются к маршрутам при помощи свойства `canActivateChild`, которому присваивается массив типов служб, реализующих метод `canActivateChild`. Этот метод будет вызван перед тем, как Angular активизирует любых потомков маршрута. В листинге 27.14 метод `canActivateChild` добавляется в класс защитника из предыдущего раздела.

Листинг 27.14. Реализация защитников дочерних маршрутов в файле terms.guard.ts

```
import { Injectable } from "@angular/core";
import {
    ActivatedRouteSnapshot, RouterStateSnapshot,
    Router
} from "@angular/router";
import { MessageService } from "./messages/message.service";
import { Message } from "./messages/message.model";

@Injectable()
export class TermsGuard {

    constructor(private messages: MessageService,
                private router: Router) { }

    canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot):
        Promise<boolean> | boolean {

        if (route.params["mode"] == "create") {

            return new Promise<boolean>((resolve, reject) => {
```

```
                let responses: [[string, (string) => void]] = [
                    ["Yes", () => { resolve(true) }],
                    ["No", () => {
                        this.router.navigateByUrl(this.router.url);
                        resolve(false);
                    }]
                ];
                this.messages.reportMessage(
                    new Message("Do you accept the terms & conditions?",
                        false, responses));
            });
        } else {
            return true;
        }
    }

    canActivateChild(route: ActivatedRouteSnapshot, state: RouterStateSnapshot):
        Promise<boolean> | boolean {

        if (route.url.length > 0
            && route.url[route.url.length - 1].path == "categories") {

            return new Promise<boolean>((resolve, reject) => {
                let responses: [[string, (string) => void]] = [
                    ["Yes", () => { resolve(true) }],
                    ["No ", () => {
                        this.router.navigateByUrl(state.url.replace("categories",
                            "products"));
                        resolve(false);
                    }]
                ];

                this.messages.reportMessage(
                    new Message("Do you want to see the categories component?",
                        false, responses));
            });
        } else {
            return true;
        }
    }
}
```

Защитник предохраняет только дочерний маршрут `categories`, а для любого другого маршрута немедленно возвращает `true`. Защитник обращается с запросом к пользователю с использованием службы сообщений, но при выборе кнопки No делает нечто иное. Кроме отклонения активного маршрута, защитник переходит на другой URL-адрес с использованием службы `Router`, которая передается в аргументе конструктора. Это стандартная схема аутентификации: пользователь перенаправляется на компонент, который затребует удостоверения безопасности при попытке выполнения защищенной операции. В нашем примере все проще: защитник переходит по маршруту того же уровня, который отображает другой компонент. (Пример использования защитников маршрутов для навигации в приложении SportsStore продемонстрирован в главе 9.)

Чтобы увидеть эффект применения защитника, щелкните на кнопке Count Categories (рис. 27.6). При выборе кнопки Yes отображается компонент `CategoryCountComponent` с количеством категорий в таблице. Кнопка No отклоняет активный маршрут и переходит по маршруту, отображающему `ProductCountComponent`.

ПРИМЕЧАНИЕ

Защитники применяются только при изменении активного маршрута. Таким образом, например, если щелкнуть на кнопке Count Categories при активном URL-адресе /table, появится запрос и кнопка Yes изменит активный маршрут. Но при повторном нажатии кнопки Count Categories ничего не произойдет, потому что Angular не инициирует изменение маршрута при совпадении целевого и активного маршрута.

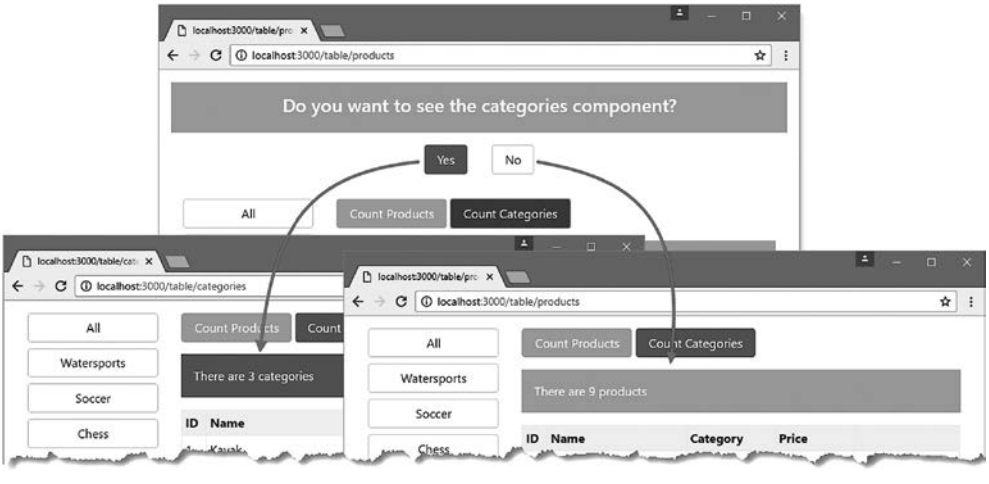


Рис. 27.6. Защита дочерних маршрутов

Предотвращение деактивизации маршрутов

Начиная работать с маршрутами, разработчик обычно концентрируется на активизации маршрутов в ответ на навигацию и на представлении нового контента для пользователя. Но не менее важную роль играет деактивизация маршрутов, которая происходит при уходе пользователя с маршрута.

Самое распространенное применение защитников деактивизации — предотвращение ухода пользователя при наличии несохраненных изменений в данных. В этом разделе мы создадим защитника, который предупреждает пользователя о возможной потере несохраненных изменений при редактировании товара. В листинге 27.15 в класс `FormComponent` вносятся изменения, упрощающие работу защитника.

Листинг 27.15. Подготовка к применению защитника в файле form.component.ts

```
import { Component, Inject } from "@angular/core";
import { NgForm } from "@angular/forms";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
import { ActivatedRoute, Router } from "@angular/router";

@Component({
    selector: "paForm",
    moduleId: module.id,
    templateUrl: "form.component.html",
    styleUrls: ["form.component.css"]
})
export class FormComponent {
    product: Product = new Product();
    originalProduct = new Product();

    constructor(private model: Model, activeRoute: ActivatedRoute,
        private router: Router) {
        activeRoute.params.subscribe(params => {
            this.editing = params["mode"] == "edit";
            let id = params["id"];
            if (id != null) {
                Object.assign(this.product, model.getProduct(id) || new Product());
                Object.assign(this.originalProduct, this.product);
            }
        })
    }

    editing: boolean = false;

    submitForm(form: NgForm) {
        if (form.valid) {
            this.model.saveProduct(this.product);
            this.originalProduct = this.product;
            this.router.navigateByUrl("/");
        }
    }

    //resetForm() {
    //    this.product = new Product();
    //}
}
```

Когда компонент начинает редактирование, он создает копию объекта `Product`, полученную от модели данных, и задает ее свойству `originalProduct`. Свойство будет использоваться защитником деактивизации для проверки наличия несохраненных изменений. Чтобы защитник не прерывал операции сохранения, перед навигационным запросом свойству `originalProduct` в методе `submitForm` присваивается объект редактируемого товара.

Соответствующее изменение также необходимо внести в шаблон, чтобы кнопка Cancel не вызывала обработчика события `reset` формы (листинг 27.16).

Листинг 27.16. Блокировка сброса формы в файле form.component.html

```
<div class="bg-primary p-a-1" [class.bg-warning]="editing">
    <h5>{{editing  ? "Edit" : "Create"}} Product</h5>
</div>
<div *ngIf="editing" class="p-t-1">
    <button class="btn btn-secondary"
        [routerLink]="['/form', 'edit', model.getPreviousProductid(product.id)]">
            Previous
    </button>
    <button class="btn btn-secondary"
        [routerLink]="['/form', 'edit', model.getNextProductId(product.id)]">
            Next
    </button>
</div>
<!--<form novalidate #form="ngForm" (ngSubmit)="submitForm(form)"
(reset)="resetForm()" >-->
<form novalidate #form="ngForm" (ngSubmit)="submitForm(form)" >
    <div class="form-group">
        <label>Name</label>
        <input class="form-control" name="name"
               [(ngModel)]="product.name" required />
    </div>

    <div class="form-group">
        <label>Category</label>
        <input class="form-control" name="category"
               [(ngModel)]="product.category" required />
    </div>

    <div class="form-group">
        <label>Price</label>
        <input class="form-control" name="price"
               [(ngModel)]="product.price"
               required pattern="^[0-9\.]+$" />
    </div>

    <button type="submit" class="btn btn-primary"
            [class.btn-warning]="editing" [disabled]="form.invalid">
        {{editing ? "Save" : "Create"}}
    </button>
    <button type="button" class="btn btn-secondary" routerLink="/">Cancel</button>
</form>
```

Чтобы создать защитника, создайте файл unsaved.guard.ts в папке exampleApp/core/app и включите в него определение класса из листинга 27.17.

Листинг 27.17. Содержимое файла unsaved.guard.ts в папке exampleApp/app/core

```
import { Injectable } from "@angular/core";
import {
    ActivatedRouteSnapshot, RouterStateSnapshot,
    Router
} from "@angular/router";
import { Observable } from "rxjs/Observable";
```

```
import { Subject } from "rxjs/Subject";
import { MessageService } from "../messages/message.service";
import { Message } from "../messages/message.model";
import { FormComponent } from "./form.component";

@Injectable()
export class UnsavedGuard {

    constructor(private messages: MessageService,
                private router: Router) { }

    canDeactivate(component: FormComponent, route: ActivatedRouteSnapshot,
        state: RouterStateSnapshot): Observable<boolean> | boolean {

        if (component.editing) {
            if (["name", "category", "price"]
                .some(prop => component.product[prop]
                    != component.originalProduct[prop])) {
                let subject = new Subject<boolean>();

                let responses: [[string, (string) => void]] = [
                    ["Yes", () => {
                        subject.next(true);
                        subject.complete();
                    }],
                    ["No", () => {
                        this.router.navigateByUrl(this.router.url);
                        subject.next(false);
                        subject.complete();
                    }]
                ];
                this.messages.reportMessage(new Message("Discard Changes?",
                    true, responses));
                return subject;
            }
        }
        return true;
    }
}
```

Защитник деактивизации определяет класс `canDeactivate`, который получает три аргумента: деактивизируемый компонент, объекты `ActivatedRouteSnapshot` и `RouteStateSnapshot`. Защитник проверяет, есть ли в компоненте несохраненные изменения, и, обнаружив их, предлагает пользователю подтвердить решение. Для разнообразия этот защитник использует объект `Observable<true>`, реализованный как `Subject<true>` (вместо `Promise<true>`), для того чтобы сообщить Angular, следует ли активизировать маршрут на основании ответа пользователя.

ПРИМЕЧАНИЕ

Обратите внимание: метод complete вызывается для Subject после вызова метода next. В противном случае Angular бесконечно долго ожидает вызова complete, что приведет к фактическому зависанию приложения.

Затем необходимо зарегистрировать защитника как службу в содержащем модуле (листинг 27.18).

Листинг 27.18. Регистрация защитника как службы в файле core.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule } from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { TableComponent } from "./table.component";
import { FormComponent } from "./form.component";
import { Subject } from "rxjs/Subject";
import { StatePipe } from "./state.pipe";
import { MessageModule } from "../messages/message.module";
import { MessageService } from "../messages/message.service";
import { Message } from "../messages/message.model";
import { Model } from "../model/repository.model";
import { RouterModule } from "@angular/router";
import { ProductCountComponent } from "./productCount.component";
import { CategoryCountComponent } from "./categoryCount.component";
import { NotFoundComponent } from "./notFound.component";
import { UnsavedGuard } from "./unsaved.guard";

@NgModule({
    imports: [BrowserModule, FormsModule, ModelModule, MessageModule,
RouterModule],
    declarations: [TableComponent, FormComponent, StatePipe,
        ProductCountComponent, CategoryCountComponent, NotFoundComponent],
    providers: [UnsavedGuard],
    exports: [ModelModule, TableComponent, FormComponent]
})
export class CoreModule { }
```

Наконец, в листинге 27.19 защитник применяется к конфигурации маршрутизации приложения. Защитники деактивизации применяются к маршрутам через свойство `canDeactivate`, которому присваивается массив служб защитников.

Листинг 27.19. Применение защитника в файле app.routing.ts

```
import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "./core/table.component";
import { FormComponent } from "./core/form.component";
import { NotFoundComponent } from "./core/notFound.component";
import { ProductCountComponent } from "./core/productCount.component";
import { CategoryCountComponent } from "./core/categoryCount.component";
import { ModelResolver } from "./model/model.resolver";
import { TermsGuard } from "./terms.guard";
import { UnsavedGuard } from "./core/unsaved.guard";

const childRoutes: Routes = [
    {
        path: "",
        canActivateChild: [TermsGuard],
```

```
        children: [{ path: "products", component: ProductCountComponent },
                   { path: "categories", component: CategoryCountComponent },
                   { path: "", component: ProductCountComponent }],
        resolve: { model: ModelResolver }
    }
];

const routes: Routes = [
    {
        path: "form/:mode/:id", component: FormComponent,
        resolve: { model: ModelResolver },
        canDeactivate: [UnsavedGuard]
    },
    {
        path: "form/:mode", component: FormComponent,
        resolve: { model: ModelResolver },
        canActivate: [TermsGuard]
    },
    { path: "table", component: TableComponent, children: childRoutes },
    { path: "table/:category", component: TableComponent, children: childRoutes },
    { path: "", redirectTo: "/table", pathMatch: "full" },
    { path: "**", component: NotFoundComponent }
]

export const routing = RouterModule.forRoot(routes);
```

Чтобы увидеть результат, щелкните на одной из кнопок Edit в таблице, отредактируйте данные в одном из текстовых полей и щелкните на кнопке Cancel, Next или Previous. Защитник предложит подтвердить решение перед тем, как разрешить Angular активизировать выбранный маршрут (рис. 27.7).

Динамическая загрузка функциональных модулей

Angular поддерживает возможность загрузки функциональных модулей тогда, когда они потребуются; этот механизм называется *динамической загрузкой*, или *отложенной загрузкой*. Он может быть полезен для функциональности, которая вряд ли понадобится всем пользователям. В следующих разделах мы создадим простой функциональный модуль и посмотрим, как настроить приложение, чтобы среда Angular загружала модуль только при переходе приложения по конкретному URL-адресу.

ПРИМЕЧАНИЕ

У динамической загрузки модулей есть как плюсы, так и минусы. Приложение становится меньше и быстрее загружается у большинства пользователей, что улучшает общие впечатления от него. С другой стороны, пользователям, которым нужна функциональность динамической загрузки, придется ждать, пока Angular получит модуль и его зависимости. Эффект может быть раздражающим, потому что пользователь понятия

не имеет, что одни функции были загружены, а другие нет. Создание динамически загружаемых модулей означает компромисс между улучшением качества взаимодействия у одних пользователей и ухудшением его у других пользователей. Подумайте, как ваши пользователи распределяются по этим группам, и постарайтесь не ухудшать впечатления у самых ценных и важных клиентов.

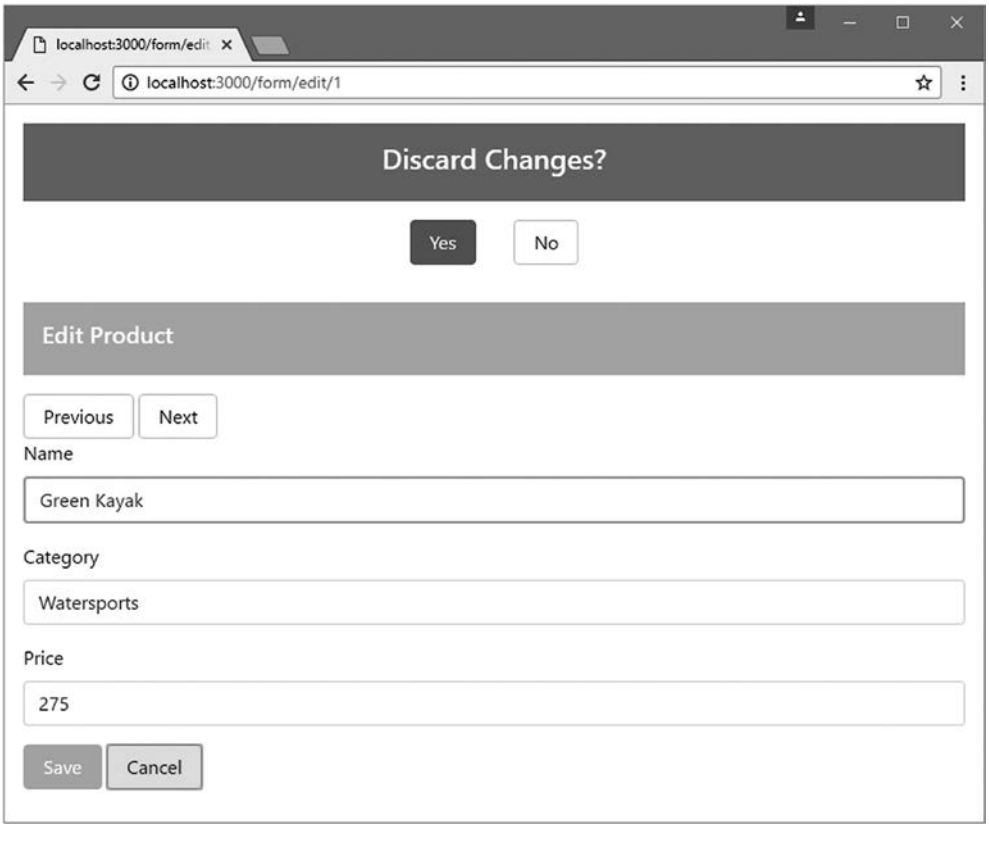


Рис. 27.7. Защита деактивизации маршрута

Создание простого функционального модуля

Динамически загружаемые модули должны содержать только ту функциональность, которая нужна лишь части пользователей. Существующие модули в нашем примере предоставляют базовую функциональность приложения, что означает, что для этой части приложения нам понадобится новый модуль. Для начала создайте папку с именем ondemand в папке exampleApp/app. Чтобы создать компонент для нового модуля, создайте файл first.component.ts в папке example/app/ondemand и добавьте код из листинга 27.20.

ВНИМАНИЕ

Очень важно не создавать зависимости между другими частями приложения и классами динамически загружаемого модуля, чтобы загрузчик модулей JavaScript не пытался загрузить модуль до того, как в нем возникнет необходимость.

Листинг 27.20. Содержимое файла ondemand.component.ts в папке exampleApp/app/ondemand

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

@Component({
    selector: "ondemand",
    moduleId: module.id,
    templateUrl: "ondemand.component.html"
})
export class OndemandComponent { }
```

Чтобы создать шаблон для компонента, создайте файл ondemand.component.html и включите в него разметку из листинга 27.21.

Листинг 27.21. Файл ondemand.component.html в папке exampleApp/app/ondemand

```
<div class="bg-primary p-a-1">This is the ondemand component</div>
<button class="btn btn-primary m-t-1" routerLink="/" >Back</button>
```

Шаблон содержит сообщение, которое явно указывает на то, что компонент был выбран, а также элемент `button`, при щелчке на котором происходит возврат к корневому URL-адресу приложения.

Чтобы определить модуль, создайте файл ondemand.module.ts и добавьте в него код из листинга 27.22.

Листинг 27.22. Содержимое файла ondemand.module.ts в папке exampleApp/app/ondemand

```
import { NgModule } from "@angular/core";
import { CommonModule } from "@angular/common";
import { OndemandComponent } from "./ondemand.component";

@NgModule({
    imports: [CommonModule],
    declarations: [OndemandComponent],
    exports: [OndemandComponent]
})
export class OndemandModule { }
```

Модуль импортирует функциональность `CommonModule`, которая используется вместо функциональности `BrowserModule` конкретного браузера для обращения к встроенным директивам в функциональных модулях, загружаемых при необходимости.

Динамическая загрузка модулей

Подготовка динамической загрузки модуля состоит из двух шагов. Первый шаг — настройка конфигурации маршрутизации в функциональном модуле для определения условия загрузки модуля. В листинге 27.23 в функциональный модуль добавляется один маршрут.

Листинг 27.23. Определение конфигурации маршрутизации в файле ondemand.module.ts

```
import { NgModule } from "@angular/core";
import { CommonModule } from "@angular/common";
import { OndemandComponent } from "./ondemand.component";
import { RouterModule } from "@angular/router";

let routing = RouterModule.forChild([
    { path: "", component: OndemandComponent }
]);

@NgModule({
    imports: [CommonModule, routing],
    declarations: [OndemandComponent],
    exports: [OndemandComponent]
})
export class OndemandModule { }
```

Маршруты в динамически загружаемых модулях определяются с использованием тех же свойств, как и в основной части приложения, и могут пользоваться всей функциональностью, включая дочерние компоненты, защитников и перенаправление. Маршрут, определяемый в листинге, соответствует пустому пути и выбирает для отображения `OndemandComponent`.

Одно важное отличие — метод, используемый для генерирования модуля, содержащего информацию маршрутизации:

```
...
let routing = RouterModule.forChild([
    { path: "", component: OndemandComponent }
]);
...
```

При создании конфигурации маршрутизации уровня приложения я использовал метод `RouterModule.forRoot`. Этот метод предназначен для определения маршрутов в корневом модуле приложения. При создании динамически загружаемых модулей следует использовать метод `RouterModule.forChild`; этот метод создает конфигурацию маршрутизации, которая подключается к общей системе маршрутизации при загрузке модуля.

Создание маршрута для динамической загрузки модуля

Второй шаг определения динамически загружаемого модуля — создание в основной части приложения маршрута, который передает Angular информацию о местонахождении модуля (листинг 27.24).

Листинг 27.24. Создание маршрута для динамической загрузки модуля в файле app.routing.ts

```
import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "./core/table.component";
import { FormComponent } from "./core/form.component";
import { NotFoundComponent } from "./core/notFound.component";
import { ProductCountComponent } from "./core/productCount.component";
import { CategoryCountComponent } from "./core/categoryCount.component";
import { ModelResolver } from "./model/model.resolver";
import { TermsGuard } from "./terms.guard";
import { UnsavedGuard } from "./core/unsaved.guard";

const childRoutes: Routes = [
    {
        path: "",
        canActivateChild: [TermsGuard],
        children: [{ path: "products", component: ProductCountComponent },
                   { path: "categories", component: CategoryCountComponent },
                   { path: "", component: ProductCountComponent }],
        resolve: { model: ModelResolver }
    }
];

const routes: Routes = [
    {
        path: "ondemand",
        loadChildren: "app/ondemand/ondemand.module#OndemandModule"
    },
    {
        path: "form/:mode/:id", component: FormComponent,
        resolve: { model: ModelResolver },
        canDeactivate: [UnsavedGuard]
    },
    {
        path: "form/:mode", component: FormComponent,
        resolve: { model: ModelResolver },
        canActivate: [TermsGuard]
    },
    { path: "table", component: TableComponent, children: childRoutes },
    { path: "table/:category", component: TableComponent, children: childRoutes },
    { path: "", redirectTo: "/table", pathMatch: "full" },
    { path: "**", component: NotFoundComponent }
]

export const routing = RouterModule.forRoot(routes);
```

Свойство `loadChildren` передает Angular информацию о том, как должен загружаться модуль. Значение свойства состоит из пути к файлу JavaScript с кодом модуля (без расширения файла), за которым следует символ `#` и имя класса модуля. Значение в листинге приказывает Angular загрузить класс `OndemandModule` из файла app/ondemand/ondemand.module.js.

Использование динамически загружаемого модуля

Остается совсем немного: добавить поддержку навигации по URL-адресу, активизирующему маршрут для динамически загружаемого модуля. Для этого мы добавим в шаблон компонента таблицы элемент `button` (листинг 27.25).

Листинг 27.25. Добавление поддержки навигации в файле table.component.html

```
<div class="col-xs-3">
    <button class="btn btn-secondary btn-block"
            routerLink="/" routerLinkActive="active"
            [routerLinkActiveOptions]="{exact: true}">
        All
    </button>
    <button *ngFor="let category of categories" class="btn btn-secondary btn-block"
            [routerLink]="['/table', category]" routerLinkActive="active">
        {{category}}
    </button>
</div>
<div class="col-xs-9">

    < !-- ...элементы опущены для краткости... -->

</div>
<div class="col-xs-12 p-t-1">
    <button class="btn btn-primary" routerLink="/form/create">
        Create New Product
    </button>
    <button class="btn btn-danger" (click)="deleteProduct(-1)">
        Generate HTTP Error
    </button>
    <button class="btn btn-danger" routerLink="/does/not/exist">
        Generate Routing Error
    </button>
    <button class="btn btn-danger" routerLink="/ondemand">
        Load Module
    </button>
</div>
```

Для выбора маршрута, загружающего модуль, никакие специальные меры не нужны, а кнопка Load Module в листинге использует стандартный атрибут `routerLink` для перехода к URL-адресу, заданному маршрутом, который был добавлен в листинге 27.24.

Чтобы увидеть, как работает динамически загружаемый модуль, используйте средства разработчика для просмотра списка файлов, загружаемых при запуске приложения. Вы не увидите запросы HTTP для файлов динамически загружаемого модуля, пока не щелкнете на кнопке Load Module. Angular использует конфигурацию маршрутизации для загрузки модуля, анализа его конфигурации маршрутизации и выбора компонента, который будет выводиться для пользователя (рис. 27.8).

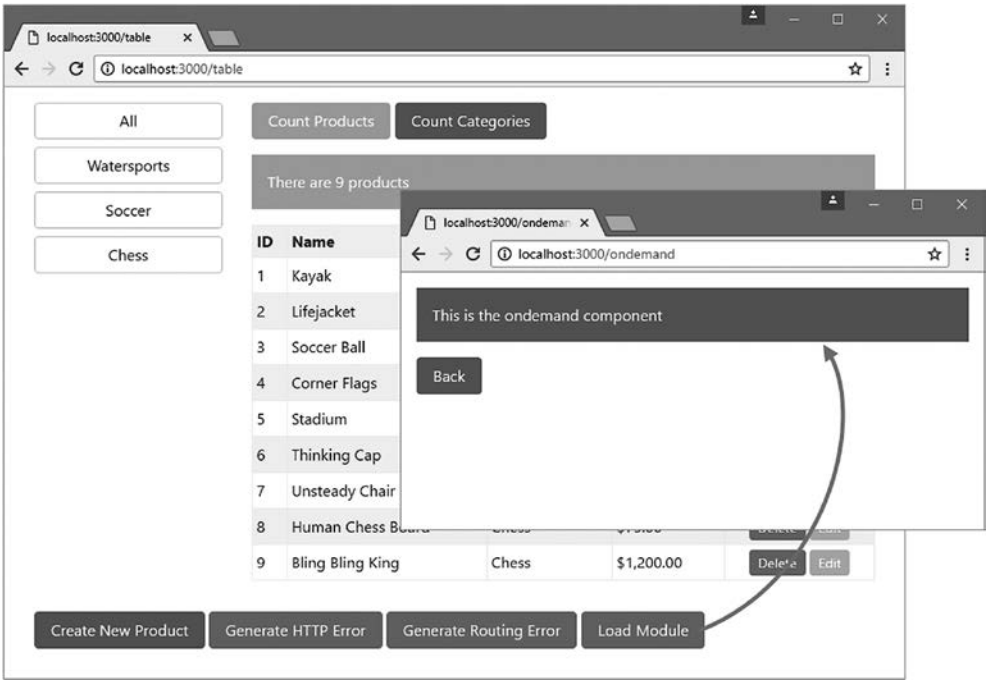


Рис. 27.8. Динамическая загрузка модуля

Защита динамических модулей

Вы можете использовать защитников с динамически загружаемыми модулями, чтобы они загружались только при нахождении приложения в конкретном состоянии или если пользователь явно согласился ожидать, пока Angular выполняет загрузку (второй вариант обычно используется только для функций администрирования, когда пользователь имеет некоторое представление о структуре приложения).

Защитник модуля должен быть определен в основной части приложения, поэтому создайте файл load.guard.ts в папке exampleApp/app и включите определение класса из листинга 27.26.

Листинг 27.26. Содержимое файла load.guard.ts в папке exampleApp/app

```
import { Injectable } from "@angular/core";
import { Route, Router } from "@angular/router";
import { MessageService } from "./messages/message.service";
import { Message } from "./messages/message.model";

@Injectable()
export class LoadGuard {
    private loaded: boolean = false;
```

```
    constructor(private messages: MessageService,
                private router: Router) { }

    canLoad(route: Route): Promise<boolean> | boolean {

        return this.loaded || new Promise<boolean>((resolve, reject) => {
            let responses: [[string, (string) => void]] = [
                ["Yes", () => {
                    this.loaded = true;
                    resolve(true);
                }],
                ["No", () => {
                    this.router.navigateByUrl(this.router.url);
                    resolve(false);
                }]
            ];

            this.messages.reportMessage(
                new Message("Do you want to load the module?",
                    false, responses));
        });
    }
}
```

Защитники динамической загрузки представляют собой классы, реализующие метод с именем `canLoad`. Этот метод вызывается, когда Angular потребуется активизировать примененный маршрут; ему передается объект `Route` с описанием маршрута.

Защитник необходим только при первой активизации URL-адреса, загружающего модуль, поэтому он определяет свойство `loaded`. Если модуль был загружен, свойству задается значение `true`, чтобы последующие запросы принимались немедленно. В противном случае защитник действует по такой же схеме, как в приведенных примерах, и возвращает объект `Promise`, который будет обработан при щелчке на одной из кнопок, отображаемых службой сообщений.

В листинге 27.27 защитник регистрируется в качестве службы в корневом модуле.

Листинг 27.27. Регистрация защитника как службы в файле app.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { ModelModule } from "./model/model.module";
import { CoreModule } from "./core/core.module";
import { TableComponent } from "./core/table.component";
import { FormComponent } from "./core/form.component";
import { MessageModule } from "./messages/message.module";
import { MessageComponent } from "./messages/message.component";
import { routing } from "./app.routing";
import { AppComponent } from "./app.component";
import { TermsGuard } from "./terms.guard"
import { LoadGuard } from "./load.guard";

@NgModule({
    imports: [BrowserModule, CoreModule, MessageModule, routing],
```

```
    declarations: [AppComponent],
    providers: [TermsGuard, LoadGuard],
    bootstrap: [AppComponent]
})
export class AppModule { }
```

Применение защитника динамической загрузки

Защитники динамической загрузки применяются к маршрутам, использующим свойство `canLoad`, в котором передается массив типов защитников. В листинге 27.28 класс `LoadGuard`, определенный в листинге 27.26, применяется к маршруту динамически загружаемого модуля.

Листинг 27.28. Защита маршрута в файле app.routing.ts

```
import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "./core/table.component";
import { FormComponent } from "./core/form.component";
import { NotFoundComponent } from "./core/notFound.component";
import { ProductCountComponent } from "./core/productCount.component";
import { CategoryCountComponent } from "./core/categoryCount.component";
import { ModelResolver } from "./model/model.resolver";
import { TermsGuard } from "./terms.guard";
import { UnsavedGuard } from "./core/unsaved.guard";
import { LoadGuard } from "./load.guard";

const childRoutes: Routes = [
    {
        path: "",
        canActivateChild: [TermsGuard],
        children: [{ path: "products", component: ProductCountComponent },
                   { path: "categories", component: CategoryCountComponent },
                   { path: "", component: ProductCountComponent }],
        resolve: { model: ModelResolver }
    }
];

const routes: Routes = [
    {
        path: "ondemand",
        loadChildren: "app/ondemand/ondemand.module#OndemandModule",
        canLoad: [LoadGuard]
    },
    {
        path: "form/:mode/:id", component: FormComponent,
        resolve: { model: ModelResolver },
        canDeactivate: [UnsavedGuard]
    },
    {
        path: "form/:mode", component: FormComponent,
        resolve: { model: ModelResolver },
        canActivate: [TermsGuard]
    },
```

```
    { path: "table", component: TableComponent, children: childRoutes },
    { path: "table/:category", component: TableComponent, children: childRoutes },
    { path: "", redirectTo: "/table", pathMatch: "full" },
    { path: "**", component: NotFoundComponent }
]

export const routing = RouterModule.forRoot(routes);
```

В результате пользователю предлагается подтвердить загрузку модуля в первый раз, когда Angular пытается активизировать маршрут (рис. 27.9).

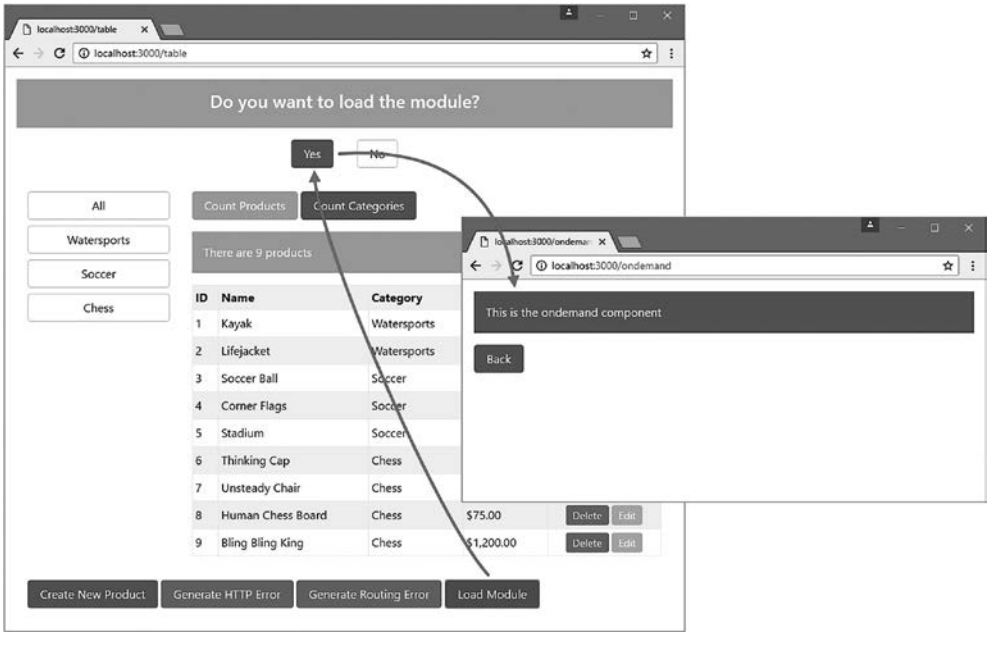


Рис. 27.9. Применение защитника динамической загрузки

Именованные элементы router-outlet

Шаблон может содержать более одного элемента `router-outlet`, что позволяет одному URL-адресу выбрать несколько компонентов, выводимых для пользователя.

Чтобы продемонстрировать эту возможность, мы добавим два новых компонента в модуль `ondemand`. Создайте файл first.component.ts в папке exampleApp/app/ondemand и включите в него определение компонента из листинга 27.29.

Листинг 27.29. Файл first.component.ts в папке exampleApp/app/ondemand

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

@Component({
    selector: "first",
```

```
    template: `<div class="bg-primary p-a-1">First Component</div>`
})
export class FirstComponent { }
```

Компонент использует встроенный шаблон для вывода сообщения, единственная цель которого — ясно показать, какой компонент был выбран системой маршрутизации. Затем создайте файл second.component.ts в папке exampleApp/app/ondemand и включите в него код компонента из листинга 27.30.

Листинг 27.30. Файл second.component.ts в папке exampleApp/app/ondemand

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

@Component({
    selector: "second",
    template: `<div class="bg-info p-a-1">Second Component</div>`
})
export class SecondComponent { }
```

Этот компонент практически идентичен компоненту из листинга 27.29; отличаются только сообщения, выводимые компонентами с использованием встроенного шаблона.

Создание дополнительных элементов router-outlet

Когда вы используете несколько элементов `router-outlet` в одном шаблоне, Angular необходимо как-то различать их. Для этого используется атрибут `name`, который позволяет однозначно идентифицировать элементы `router-outlet` (листинг 27.31).

Листинг 27.31. Добавление именованных элементов router-outlet в файле ondemand.component.html

```
<div class="bg-primary p-a-1">This is the ondemand component</div>
<div class="col-xs-12 m-t-1">
    <router-outlet></router-outlet>
</div>
<div class="col-xs-6">
    <router-outlet name="left"></router-outlet>
</div>
<div class="col-xs-6">
    <router-outlet name="right"></router-outlet>
</div>
<button class="btn btn-primary m-t-1" routerLink="/" >Back</button>
```

Новая разметка создает три новых элемента `router-outlet`. Шаблон может содержать не более одного элемента `router-outlet` без атрибута `name`; этот элемент называется *первичным*. Это связано с тем, что отсутствие атрибута `name` равносильно его применению со значением `primary`. Все элементы маршрутизации, встречавшиеся в книге до настоящего момента, использовали первичный элемент `router-outlet` для отображения компонентов.

Каждый элемент `router-outlet` должен содержать элемент `name` с уникальным именем. В листинге использовались имена `left` и `right`, потому что классы, назначаемые элементам `div`, содержащим `router-outlet`, используют CSS для размещения этих двух элементов рядом друг с другом.

На следующем шаге создается маршрут, включающий информацию о том, какой компонент должен выводиться в каждом из элементов `router-outlet` (листинг 27.32). Если Angular не может найти маршрут, соответствующий какому-то элементу `router-outlet`, то в этом элементе контент не выводится.

Листинг 27.32. Использование именованных элементов router-outlet в файле ondemand.module.ts

```
import { NgModule } from "@angular/core";
import { CommonModule } from "@angular/common";
import { OndemandComponent } from "./ondemand.component";
import { RouterModule } from "@angular/router";
import { FirstComponent } from "./first.component";
import { SecondComponent } from "./second.component";

let routing = RouterModule.forChild([
    {
        path: "",
        component: OndemandComponent,
        children: [
            { path: "",
              children: [
                   { outlet: "primary", path: "", component: FirstComponent, },
                   { outlet: "left", path: "", component: SecondComponent, },
                   { outlet: "right", path: "", component: SecondComponent, },
              ]},
        ]
    },
]);

@NgModule({
    imports: [CommonModule, routing],
    declarations: [OndemandComponent, FirstComponent, SecondComponent],
    exports: [OndemandComponent]
})
export class OndemandModule { }
```

Свойство `outlet` используется для назначения элемента `router-outlet`, к которому относится маршрут. Конфигурация маршрутизации в листинге соответствует пустому пути для всех трех элементов `router-outlet`, и для них выбираются только что созданные компоненты: первичный элемент `router-outlet` будет отображать `FirstComponent`, а элементы `left` и `right` — `SecondComponent` (рис. 27.10). Чтобы увидеть эффект, щелкните на кнопке Load Module, а потом по запросу щелкните на кнопке Yes.

ПРИМЕЧАНИЕ

Если свойство outlet не указано, то Angular считает, что маршрут относится к первичному элементу router-outlet. Я предпочитаю включать свойство outlet во все маршруты, чтобы было сразу видно, какому элементу router-outlet соответствует маршрут.

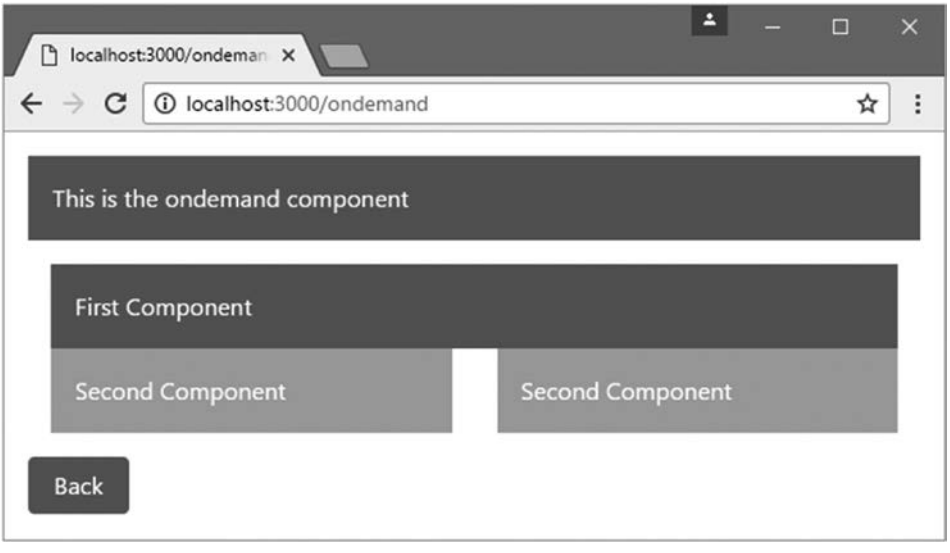


Рис. 27.10. Использование нескольких элементов router-outlet

При активизации маршрута Angular ищет соответствия для каждого элемента `router-outlet`. Маршруты всех трех новых элементов `router-outlet` соответствуют пустому пути, что позволяет Angular отобразить компоненты, изображенные на иллюстрации.

Навигация при использовании нескольких элементов router-outlet

Изменение компонентов, отображаемых каждым элементом `router-outlet`, означает создание нового набора маршрутов и переход по содержащему их URL-адресу. В листинге 27.33 создается маршрут, который соответствует пути `/ondemand/swap` и который переключает компоненты, отображаемые тремя элементами `router-outlet`.

УПРОЩЕНИЕ ИМЕНОВАННЫХ ЭЛЕМЕНТОВ ROUTER-OUTLET

Способ создания маршрутов и навигации в приложении, продемонстрированный в этом разделе, не единственный, но он надежно работает и может использоваться в динамически загружаемых модулях.

Альтернативный метод основан на навигации по URL-адресам, содержащим пути всех целевых элементов router-outlet, как в следующем примере:

http://localhost:3000/ondemand/primarypath(left:leftpath//right:rightpath)

Основная часть URL-адреса выбирает первичный элемент router-outlet, а части в скобках — элементы left и right. Проблема в том, что в Angular поиск маршрутов для име-

нованных элементов router-outlet реализован непоследовательно и вы легко можете создать конфигурацию маршрутизации, которая работает не так, как задумано, или не работает вообще.

В использованном решении есть один скользкий момент: я выбираю бескомпонентные маршруты, у которых есть потомки, занимающиеся именованными элементами router-outlet. Это усложняет конфигурацию маршрутизации и требует создания наборов маршрутов для каждой комбинации компонентов и элементов router-outlet, которая вам понадобится. Но с другой стороны, такое решение получается более понятным и надежным даже при использовании динамически загружаемых модулей.

Листинг 27.33. Создание маршрутов для именованных элементов router-outlet в файле ondemand.module.ts

```
import { NgModule } from "@angular/core";
import { CommonModule } from "@angular/common";
import { OndemandComponent } from "./ondemand.component";
import { RouterModule } from "@angular/router";
import { FirstComponent } from "./first.component";
import { SecondComponent } from "./second.component";

let routing = RouterModule.forChild([
    {
        path: "",
        component: OndemandComponent,
        children: [
            {
                path: "",
                children: [
                    { outlet: "primary", path: "", component: FirstComponent, },
                    { outlet: "left", path: "", component: SecondComponent, },
                    { outlet: "right", path: "", component: SecondComponent, },
                ]
            },
            {
                path: "swap",
                children: [
                    { outlet: "primary", path: "", component: SecondComponent, },
                    { outlet: "left", path: "", component: FirstComponent, },
                    { outlet: "right", path: "", component: FirstComponent, },
                ]
            },
        ]
    },
]);

@NgModule({
    imports: [CommonModule, routing],
    declarations: [OndemandComponent, FirstComponent, SecondComponent],
    exports: [OndemandComponent]
})
export class OndemandModule { }
```

В листинге 27.34 в шаблон компонента добавляются элементы `button` для перехода по двум наборам маршрутов в листинге 27.33, с чередованием набора отображаемых компонентов.

Листинг 27.34. Навигация с несколькими элементами router-outlet в файле ondemand.component.html

```
<div class="bg-primary p-a-1">This is the ondemand component</div>
<div class="col-xs-12 m-t-1">
    <router-outlet></router-outlet>
</div>
<div class="col-xs-6">
    <router-outlet name="left"></router-outlet>
</div>
<div class="col-xs-6">
    <router-outlet name="right"></router-outlet>
</div>
<button class="btn btn-secondary m-t-1" routerLink="/ondemand">Normal</button>
<button class="btn btn-secondary m-t-1" routerLink="/ondemand/swap">Swap</button>
<button class="btn btn-primary m-t-1" routerLink="/">Back</button>
```

В результате кнопки Swap и Normal выполняют навигацию по маршрутам, дочерние маршруты которых сообщают Angular, какие компоненты должны отображаться каждым из элементов `router-outlet` (рис. 27.11).

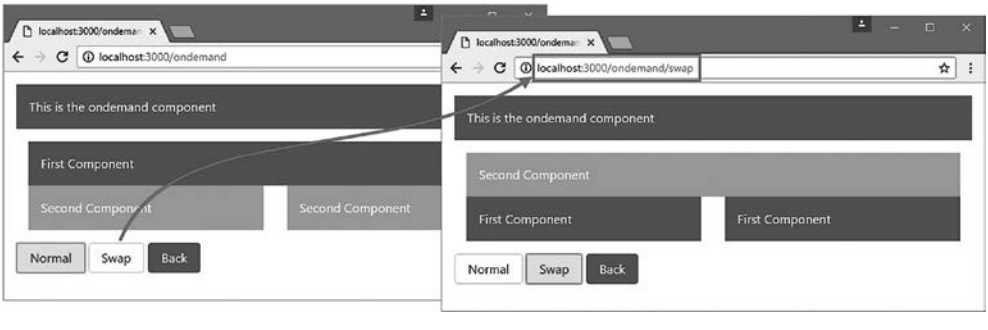


Рис. 27.11. Использование навигации для выбора нескольких элементов router-outlet

Итоги

В этой главе я завершил описание системы маршрутизации URL в Angular. Вы узнали, как назначать защитников, чтобы управлять активизацией маршрутов, как загружать модули только в тот момент, когда в них возникнет необходимость, и как использовать множественные элементы `router-outlet` для отображения компонентов. В следующей главе вы научитесь использовать анимацию в приложениях Angular.

28 Анимация

В этой главе я опишу систему анимации Angular, использующую привязки данных для анимации элементов HTML в соответствии с состоянием приложения. В широком смысле анимации решают две задачи в приложениях Angular: они подчеркивают изменения в контенте или сглаживают их.

Первое важно в том случае, если изменения в контенте могут быть не очевидны для пользователя. В нашем примере кнопки Previous и Next при редактировании продукта изменяют поля данных, но не создают других визуальных изменений; их нажатие может привести к изменению состояния, которое останется незаметным для пользователя. Анимации помогут привлечь внимание к таким изменениям, и пользователю будет проще заметить результат действия.

Сглаживание изменений делает работу с приложением более приятной. Когда пользователь щелкает на кнопке Edit, чтобы начать редактирование товара, переключение контента в приложении может быть слишком неожиданным. Использование анимаций для замедления перехода создает ощущение контекста при контентном изменении и делает его менее резким. В этой главе я покажу, как работает система анимации и как она используется для привлечения (или отвлечения) внимания при внезапных переходах.

ПОДДЕРЖКА АНИМАЦИИ В БРАУЗЕРАХ

Средства анимации Angular работают на основе API Web Animations, надежно реализованного только в Chrome и Mozilla. Другие браузеры либо вообще не поддерживают API, либо не реализуют все его возможности. Я использую библиотеку полизаполнения для реализации базовой поддержки в старых браузерах, но некоторые функции все равно работать не будут. Возможно, со временем ситуация улучшится, но вы должны тщательно провести тестирование и убедиться в том, что браузеры, на которые вы ориентируетесь в своем проекте, поддерживают используемые средства анимации.

Если вы хотите воспроизвести приведенные примеры и получить ожидаемые результаты, используйте Chrome.

В табл. 28.1 средства анимации Angular представлены в контексте.

Таблица 28.1. Анимация Angular в контексте

Вопрос	Ответ
Что это такое?	Система анимации может изменять внешний вид элементов HTML в соответствии с изменениями в состоянии приложения
Для чего они нужны?	При разумном использовании анимация делает приложение более приятным в использовании
Как они используются?	Анимации определяются при помощи функций, определенных в модуле @angular/core, регистрируются при помощи свойства декоратора @Component и применяются в привязке данных
Есть ли у них недостатки или скрытые проблемы?	Полноценная поддержка анимаций реализована лишь в немногих браузерах, соответственно вы не можете рассчитывать на их нормальную работу в поддержке Angular, реализованной для других браузеров
Есть ли альтернативы?	Единственная альтернатива — отказ от использования анимаций

В табл. 28.2 приведена краткая сводка материала главы.

Таблица 28.2. Сводка материала главы

Проблема	Решение	Листинг
Привлечение внимания пользователя к переходу в состояние элемента	Примените анимацию	1–9
Анимация перехода элемента из одного состояния в другое	Используйте переход	10–15
Параллельное выполнение анимации	Используйте группы анимации	16
Использование одних и тех же стилей в нескольких анимациях	Используйте общие стили	17
Анимация позиции элемента	Используйте преобразование элемента	18
Использование анимаций для применения стилей CSS	Используйте DOM и CSS API	19, 20
Получение уведомлений об анимациях	Получайте события анимации	21, 22

Подготовка проекта

В этой главе мы продолжим работать с проектом `exampleApp`, который был создан в главе 22 и использовался во всех последующих главах. Ниже рассказано, как подготовить приложение для применения функциональности, которой посвящена эта глава.

Добавление полизаполнения анимации

Поддержка автоматизации в Angular зависит от JavaScript API, поддерживаемого только последними версиями браузеров, и для работы анимации в старых браузерах потребуется полизаполнение. В листинге 28.1 это полизаполнение добавляется в набор пакетов приложения.

Листинг 28.1. Добавление полизаполнения анимации в файл package.json

```
{
  "dependencies": {
    "@angular/common": "2.2.0",
    "@angular/compiler": "2.2.0",
    "@angular/core": "2.2.0",
    "@angular/platform-browser": "2.2.0",
    "@angular/platform-browser-dynamic": "2.2.0",
    "@angular/upgrade": "2.2.0",
    "@angular/forms": "2.2.0",
    "@angular/http": "2.2.0",
    "@angular/router": "3.2.0",
    "reflect-metadata": "0.1.8",
    "rxjs": "5.0.0-beta.12",
    "zone.js": "0.6.26",
    "core-js": "2.4.1",
    "classlist.js": "1.1.20150312",
    "systemjs": "0.19.40",
    "bootstrap": "4.0.0-alpha.4",
    "intl": "1.2.4",
    "html5-history-api": "4.2.7",
    "web-animations-js": "2.2.2"
  },

  "devDependencies": {
    "lite-server": "2.2.2",
    "typescript": "2.0.2",
    "typings": "1.3.2",
    "concurrently": "2.2.0",
    "systemjs-builder": "0.15.32",
    "json-server": "0.8.21"
  },

  "scripts": {
    "start": "concurrently \"npm run tscwatch\" \"npm run lite\" \"npm run json\"
",
    "tsc": "tsc",
    "tscwatch": "tsc -w",
    "lite": "lite-server",
    "json": "json-server --p 3500 restData.js",
    "typings": "typings",
    "postinstall": "typings install"
  }
}
```

Группа Angular рекомендует использовать пакет `web-animations-js` для поддержки анимации в старых браузерах. Выполните следующую команду из папки exampleApp, чтобы загрузить пакет полизаполнения:

```
npm install
```

В листинге 28.2 в файл index.html добавляется элемент `script` для файла полизаполнения.

Листинг 28.2. Добавление полизаполнения анимации в файле index.html

```
<!DOCTYPE html>
<html>
<head>
    <base href="/">
    <title></title>
    <meta charset="utf-8" />
    <script src="node_modules/html5-history-api/history.min.js"></script>
    <script src="node_modules/classlist.js/classList.min.js"></script>
    <script src="node_modules/web-animations-js/web-animations.min.js"></script>
    <script src="node_modules/core-js/client/shim.min.js"></script>
    <script src="node_modules/intl/dist/Intl.complete.js"></script>
    <script src="node_modules/zone.js/dist/zone.min.js"></script>
    <script src="node_modules/reflect-metadata/Reflect.js"></script>
    <script src="node_modules/systemjs/dist/system.src.js"></script>
    <script src="rxjs.module.min.js"></script>
    <script src="systemjs.config.js"></script>
    <script>
      System.import("app/main").catch(function(err){ console.error(err); });
    </script>
    <link href="node_modules/bootstrap/dist/css/bootstrap.min.css"
        rel="stylesheet" />
</head>
<body class="m-a-1">
    <app></app>
</body>
</html>
```

Отключение задержки HTTP

Следующим подготовительным шагом для этой главы станет отключение задержки, добавляемой к асинхронным запросам HTTP (листинг 28.3).

ПРИМЕЧАНИЕ

Все проекты для каждой главы книги можно бесплатно загрузить в составе архива, прилагаемого к книге, на сайте apress.com.

Листинг 28.3. Отключение задержки в файле rest.datasource.ts

```
import { Injectable, Inject, OpaqueToken } from "@angular/core";
import { Http, Request, RequestMethod, Headers, Response } from "@angular/http";
import { Observable } from "rxjs/Observable";
import { Product } from "./product.model";
import "rxjs/add/operator/map";
import "rxjs/add/operator/catch";
import "rxjs/add/observable/throw";
import "rxjs/add/operator/delay";

export const REST_URL = new OpaqueToken("rest_url");

@Injectable()
export class RestDataSource {

    constructor(private http: Http,
        @Inject(REST_URL) private url: string) { }

    // ...другие методы опущены для краткости...

    private sendRequest(verb: RequestMethod,
        url: string, body?: Product): Observable<Product> {

        let headers = new Headers();
        headers.set("Access-Key", "<secret>");
        headers.set("Application-Names", ["exampleApp", "proAngular"]);

        return this.http.request(new Request({
            method: verb,
            url: url,
            body: body,
            headers: headers
        }))
            //.delay(5000)
            .map(response => response.json())
            .catch((error: Response) => Observable.throw(
                `Network Error: ${error.statusText} (${error.status})`));
    }
}
```

Упрощение шаблона таблицы и конфигурации маршрутизации

Многие примеры этой главы применяются к элементам таблицы товаров. На последней стадии подготовки этой главы мы упростим шаблон компонента таблицы, чтобы сосредоточиться на меньшем объеме контента в листингах.

В листинге 28.4 показан упрощенный шаблон, в котором удалены кнопки для генерирования ошибок маршрутизации, а также кнопка и элемент `router-outlet` для счетчика категорий товаров. В листинге также удаляются кнопки для фильтрации таблицы по категориям.

Листинг 28.4. Упрощение шаблона в файле table.component.html

```
<table class="table table-sm table-bordered table-striped">
    <tr>
        <th>ID</th><th>Name</th><th>Category</th><th>Price</th><th></th>
    </tr>
    <tr *ngFor="let item of getProducts()">
        <td style="vertical-align:middle">{{item.id}}</td>
        <td style="vertical-align:middle">{{item.name}}</td>
        <td style="vertical-align:middle">{{item.category}}</td>
        <td style="vertical-align:middle">
            {{item.price | currency:"USD":true }}
        </td>
        <td class="text-xs-center">
            <button class="btn btn-danger btn-sm"
                    (click)="deleteProduct(item.id)">
                Delete
            </button>
            <button class="btn btn-warning btn-sm"
                [routerLink]="['/form', 'edit', item.id]">
                Edit
            </button>
        </td>
    </tr>
</table>

<div class="col-xs-12 p-t-1">
    <button class="btn btn-primary" routerLink="/form/create">
        Create New Product
    </button>
</div>
```

Листинг 28.5 обновляет конфигурацию маршрутизации URL, чтобы маршруты не связывались с элементом `router-outlet`, исключенным из шаблона компонента таблицы.

Листинг 28.5. Обновление конфигурации маршрутизации в файле app.routing.ts

```
import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "./core/table.component";
import { FormComponent } from "./core/form.component";
import { NotFoundComponent } from "./core/notFound.component";
import { ProductCountComponent } from "./core/productCount.component";
import { CategoryCountComponent } from "./core/categoryCount.component";
import { ModelResolver } from "./model/model.resolver";
import { TermsGuard } from "./terms.guard";
import { UnsavedGuard } from "./core/unsaved.guard";
import { LoadGuard } from "./load.guard";

const routes: Routes = [
    {
        path: "form/:mode/:id", component: FormComponent,
        canDeactivate: [UnsavedGuard]
    },
    { path: "form/:mode", component: FormComponent, canActivate: [TermsGuard] },
```

```
    { path: "table", component: TableComponent },
    { path: "table/:category", component: TableComponent },
    { path: "", redirectTo: "/table", pathMatch: "full" },
    { path: "**", component: NotFoundComponent }
]

export const routing = RouterModule.forRoot(routes);
```

Выполните следующую команду из папки exampleApp, чтобы запустить компилятор TypeScript, сервер HTTP для разработки и REST-совместимую веб-службу:

```
npm start
```

Открывается новое окно (или вкладка) браузера с контентом, показанным на рис. 28.1.

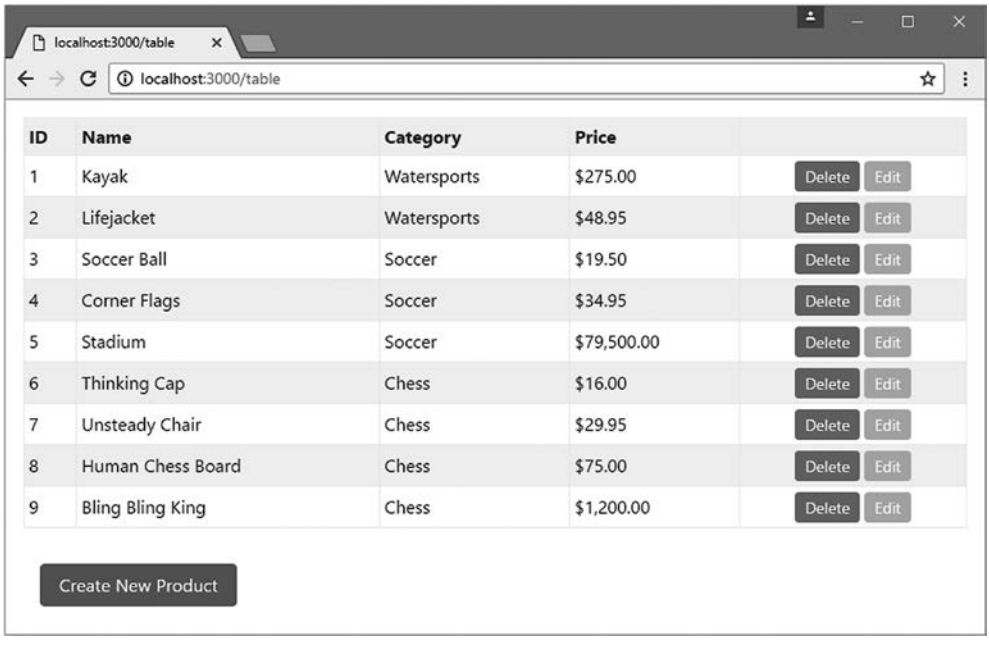


Рис. 28.1. Запуск приложения

Основы работы с анимацией в Angular

Как это часто бывает при изучении Angular, начать лучше всего с примера, который поможет мне объяснить, как работает анимация и как она сочетается с другими возможностями Angular. Сейчас мы создадим несложную анимацию, которая воздействует на строки таблицы товаров. А когда вы поймете, как работает базовая функциональность, мы рассмотрим все остальные варианты конфигурации и более подробно разберемся в том, как они работают.

Для начала мы добавим в приложение элемент `select` для выбора категории. Когда пользователь выбирает категорию, строки товаров этой категории в таблице отображаются в одном из двух стилей (табл. 28.3).

Таблица 28.3. Стили для примера использования анимации

Описание	Стиль
Товар принадлежит к выбранной категории	Строка таблицы выводится увеличенным текстом на зеленом фоне
Товар не принадлежит к выбранной категории	Строка таблицы выводится уменьшенным текстом на красном фоне

Создание анимации

Создайте файл table.animations.ts в папке example/app/core и включите в него код из листинга 28.6.

Листинг 28.6. Содержимое файла table.animations.ts в папке exampleApp/app/core

```
import { trigger, style, state, transition, animate } from "@angular/core";

export const HighlightTrigger = trigger("rowHighlight", [
    state("selected", style({
        backgroundColor: "lightgreen",
        fontSize: "20px"
    })),
    state("notselected", style({
        backgroundColor: "lightsalmon",
        fontSize: "12px"
    })),
    transition("selected => notselected", animate("200 ms")),
    transition("notselected => selected", animate("400 ms"))
]);
```

Синтаксис определения анимаций бывает весьма лаконичным, и в нем используются функции, определенные в модуле `@angular/core`. В следующих разделах мы начнем с основ и будем понемногу углубляться в подробности при рассмотрении всех структурных блоков анимации, использованных в листинге.

ПРИМЕЧАНИЕ

Не беспокойтесь, если какие-то структурные блоки, описанные в следующих разделах, покажутся непонятными. Эта область функциональности начнет проясняться только тогда, когда вы начнете понимать, как сочетаются разные части.

Определение групп стилей

В системе анимации центральное место занимает группа стилей — множество свойств стилей CSS и значений, которые будут применяться к элементам HTML. Группы стилей определяются с использованием функции `style`, которая полу-

чает объектный литерал JavaScript, определяющий соответствие между именами свойств и значений:

```
...
style({
    backgroundColor: "lightgreen",
    fontSize: "20px"
})
...
```

Группа стилей приказывает Angular назначить светло-зеленый цвет фона и выбрать для вывода текста шрифт размером 20 пикселов.

ИМЕНА СВОЙСТВ CSS

При использовании функции style свойства CSS могут задаваться двумя способами. Вы можете использовать соглашения об именах свойств JavaScript: например, свойство цвета фона элемента задается в виде backgroundColor (запись подряд, без дефисов, все слова, кроме первого, начинаются с прописной буквы). Эта схема используется в листинге 28.6:

```
...
style({
    backgroundColor: "lightgreen",
    fontSize: "20px"
})),
...
```

Также возможно использовать схему имен CSS, в которой имена свойств записываются в форме background-color (все слова начинаются со строчной буквы и разделяются дефисами). Если вы используете формат CSS, имена свойств должны заключаться в кавычки, чтобы JavaScript не пытался интерпретировать дефисы как арифметические операторы:

```
...
state("green", style({
    "background-color": "lightgreen",
    "font-size": "20px"
})),
...
```

Какую бы схему имен вы ни использовали, важно действовать последовательно. На момент написания книги Angular некорректно применяет стили при смешанном использовании схем имен свойств. Чтобы получить последовательные результаты, выберите одну схему имен и используйте ее для всех стилевых свойств в своем приложении.

Определение состояний элементов

Среда Angular должна знать, когда к элементу нужно применить набор стилей. Для этого нужно определить состояние элемента — имя, по которому вы обращаетесь к набору стилей. Состояния элементов создаются функцией `state`, которая получает имя и набор стилей, которые должны быть с ним связаны. В листинге 28.6 определяется одно из двух состояний элементов:

```
...
state("selected", style({
    backgroundColor: "lightgreen",
    fontSize: "20px"
})),
...
```

В листинге определяются два состояния, `selected` и `notselected`: они применяются к строкам таблицы в зависимости от того, принадлежит товар к категории, выбранной пользователем, или нет.

Определение переходов состояний

Если элемент HTML находится в одном из состояний, созданных функцией `state`, Angular применяет свойства CSS, входящие в группу стилей состояния. Функция `transition` сообщает Angular, как должны применяться новые свойства CSS. В листинге 28.6 определяются два перехода.

```
...
transition("selected => notselected", animate("200 ms")),
transition("notselected => selected", animate("400 ms"))
...
```

Первый аргумент, передаваемый функции `transition`, сообщает Angular, к каким состояниям применяется данная инструкция. Аргумент представляет собой строку с двумя состояниями и стрелкой, выражающей отношения между ними. Существуют две разновидности стрелок (табл. 28.4).

Таблица 28.4. Разновидности стрелок переходов анимации

Стрелка	Пример	Описание
=>	selected => notselected	Односторонний переход между двумя состояниями: элемент переходит из состояния selected в состояние notselected
<=>	selected <=> notselected	Двусторонний переход между двумя состояниями: элемент переходит из состояния selected в состояние notselected, а из состояния notselected в состояние selected

Переходы, определенные в листинге 28.6, используют односторонние стрелки: они сообщают Angular, как реагировать на переход элемента из состояния `selected` в состояние `notselected`, но не из состояния `notselected` в состояние `selected`.

Второй аргумент функции `transition` сообщает Angular, какое действие следует выполнять при изменении состояния. Функция `animate` приказывает Angular выполнить переход между свойствами набора стилей CSS, определяемого двумя состояниями элементов. Аргументы, передаваемые функции `animate` в листинге 28.6, определяют период времени, в течение которого должен происходить этот постепенный переход (200 или 400 миллисекунд).

СОВЕТЫ ПО ПРИМЕНЕНИЮ АНИМАЦИИ

Разработчики часто увлекаются применением анимации; такие приложения только раздражают пользователей. Не злоупотребляйте анимациями, они должны быть простыми и быстрыми. Анимации помогают пользователю понять, как работает ваше приложение, а не демонстрируют ваш творческий талант. Пользователям — особенно в корпоративных бизнес-приложениях — приходится многократно выполнять одни и те же задачи, поэтому излишние и долгие анимации будут только мешать.

Я и сам склонен увлекаться; без должного контроля мои приложения напоминали бы игровые автоматы в Лас-Вегасе. Чтобы держать проблему под контролем, я руководствуюсь двумя правилами. Во-первых, я выполняю основные операции в приложении по двадцать раз подряд. В нашем примере это означает создание 20 товаров с последующим редактированием 20 товаров. Я удаляю или сокращаю любые анимации, завершение которых тормозит переход к следующему шагу процесса.

Второе правило — я не отключаю анимации во время разработки. Идея закомментировать анимацию во время работы над некоторой функцией приложения выглядит соблазнительно, потому что я буду проводить серию быстрых тестов во время написания кода. Однако любая анимация, которая мешает мне, также будет мешать и пользователю, поэтому я оставляю анимации на месте и настраиваю их (обычно сокращая их продолжительность), пока они не станут менее назойливыми и раздражающими.

Разумеется, вы не обязаны соблюдать мои правила. Тем не менее важно позаботиться о том, чтобы анимации помогали пользователю, а не становились препятствием для быстрой работы или помехой, отвлекающей внимание.

Определение триггера

Последний фрагмент служебного кода — триггер анимации — упаковывает состояния элементов и переходы и назначает имя, которое может использоваться для применения анимации к компоненту. Триггеры создаются функцией `trigger`:

```
...
export const HighlightTrigger = trigger("rowHighlight", [...])
...
```

В первом аргументе передается имя триггера (`rowHighlight` в данном примере), а во втором — массив состояний и переходов, которые будут доступны при применении триггера.

Применение анимации

После того как анимация будет определена, ее можно применить к одному или нескольким компонентам при помощи свойства `animations` декоратора `@Component`. В листинге 28.7 анимация, определяемая в листинге 28.6, применяется к компоненту; при этом добавляются некоторые дополнительные функции, необходимые для поддержки анимации.

Листинг 28.7. Применение анимации в файле table.component.ts

```
import { Component, Inject } from "@angular/core";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
import { ActivatedRoute } from "@angular/router";
import { HighlightTrigger } from "./table.animations";

@Component({
    selector: "paTable",
    moduleId: module.id,
    templateUrl: "table.component.html",
    animations: [HighlightTrigger]
})
export class TableComponent {
    category: string = null;

    constructor(private model: Model, activeRoute: ActivatedRoute) {
        activeRoute.params.subscribe(params => {
            this.category = params["category"] || null;
        })
    }

    getProduct(key: number): Product {
        return this.model.getProduct(key);
    }

    getProducts(): Product[] {
        return this.model.getProducts()
            .filter(p => this.category == null || p.category == this.category);
    }

    get categories(): string[] {
        return this.model.getProducts()
            .map(p => p.category)
            .filter((category, index, array) => array.indexOf(category) == index);
    }

    deleteProduct(key: number) {
        this.model.deleteProduct(key);
    }

    highlightCategory: string = "";

    getRowState(category: string): string {
        return this.highlightCategory == "" ? "" :
            this.highlightCategory == category ? "selected" : "notselected";
    }
}
```

Свойству `animations` присваивается массив триггеров. Анимации можно определять «на месте», но они быстро усложняются, а весь компонент начинает плохо читаться; вот почему я использовал отдельный файл и экспортировал из него константное значение, которое затем задается свойству `animations`.

Другие изменения должны предоставить соответствие между категорией, выбранной пользователем, и состоянием анимации, которое будет присваиваться элементам. Значение свойства `highlightCategory` будет задаваться при помощи элемента `select` и использоваться в методе `getRowState`, где оно сообщает Angular, какое из состояний анимации в листинге 28.7 должно назначаться в зависимости от категории товара. Если товар относится к выбранной категории, то метод возвращает `selected`; в противном случае возвращается `notselected`. Если пользователь не выбрал категорию, то возвращается пустая строка.

Затем остается применить анимацию к шаблону компонента, сообщая Angular, к каким элементам должна применяться анимация (листинг 28.8). В листинге также добавляется элемент `select`, который задает значение свойства `highlightCategory` компонента, для чего используется привязка `ngModel`.

Листинг 28.8. Применение анимации в файле table.component.html

```
<div class="form-group bg-info p-a-1">
    <label>Category</label>
    <select [(ngModel)]="highlightCategory" class="form-control">
        <option value="">None</option>
        <option *ngFor="let category of categories">
            {{category}}
        </option>
    </select>
</div>
<table class="table table-sm table-bordered table-striped">
    <tr>
        <th>ID</th><th>Name</th><th>Category</th><th>Price</th><th></th>
    </tr>
    <tr *ngFor="let item of getProducts()"
            [@rowHighlight]="getRowState(item.category)">
        <td style="vertical-align:middle">{{item.id}}</td>
        <td style="vertical-align:middle">{{item.name}}</td>
        <td style="vertical-align:middle">{{item.category}}</td>
        <td style="vertical-align:middle">
            {{item.price | currency:"USD":true }}
        </td>
        <td class="text-xs-center">
            <button class="btn btn-danger btn-sm"
                    (click)="deleteProduct(item.id)">
                Delete
            </button>
            <button class="btn btn-warning btn-sm"
                [routerLink]="['/form', 'edit', item.id]">
                Edit
            </button>
        </td>
    </tr>
</table>
<div class="col-xs-12 p-t-1">
    <button class="btn btn-primary" routerLink="/form/create">
        Create New Product
    </button>
</div>
```

Анимации применяются к шаблонам при помощи специальных привязок данных, которые связывают триггер анимации с элементом HTML. Цель привязки сообщает Angular, какой триггер следует применить, а выражение привязки — как определить, какое состояние должно быть назначено элементу:

```
...
<tr *ngFor="let item of getProducts()" [@rowHighlight]="getRowState(item.
                                        category)">
...
```

Целью привязки является имя триггера анимации, снабженное префиксом `@`, обозначающим привязку. Эта привязка сообщает Angular, что триггер `rowHighlight` должен быть применен к элементу `tr`. Выражение указывает, что для определения состояния следует вызвать метод `getRowState` компонента, используя значение `item.category` как аргумент.

На рис. 28.2 изображена структура привязки данных анимации.

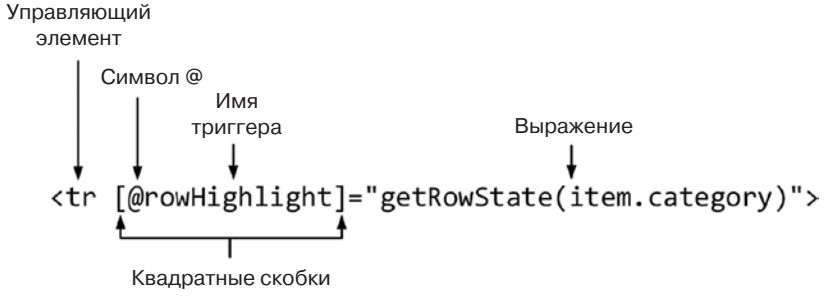


Рис. 28.2. Строение привязки данных анимации

Тестирование эффекта анимации

Изменения из предыдущего раздела добавляют над таблицей товаров элемент `select`. Чтобы понаблюдать за эффектом анимации, выберите в списке категорию `Soccer`; Angular при помощи триггера определяет, какие состояния анимации должны быть назначены каждому элементу. Строкам таблицы для товаров из категории `Soccer` будет назначено состояние `selected`, тогда как другим строкам будет назначено состояние `notselected`. Результат показан на рис. 28.3.

Новые стили применяются мгновенно. Чтобы увидеть более плавный переход, выберите в списке категорию `Chess`; вы увидите плавную анимацию при назначении строкам `Chess` состояния `selected`, а другим строкам — состояния `notselected`. Это происходит из-за того, что триггер анимации содержит переходы между этими состояниями, из-за которых Angular сопровождает изменения стилей CSS анимацией (рис. 28.4). В предыдущем изменении переход отсутствует, поэтому Angular по умолчанию применяет новые стили немедленно.

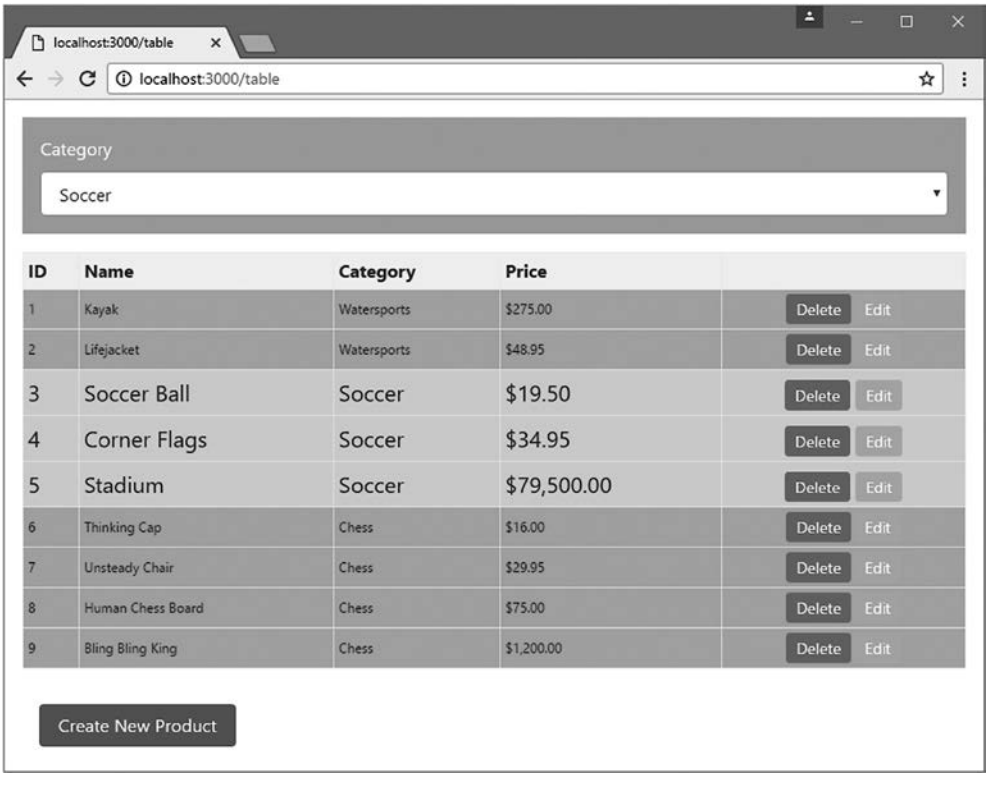


Рис. 28.3. Выбор категории товаров

ПРИМЕЧАНИЕ

Отразить эффект анимации на серии иллюстраций невозможно. Лучшее, что я могу, — изобразить некоторые промежуточные состояния. Чтобы разобраться в этой функциональности, придется поэкспериментировать. Я рекомендую загрузить проект этой главы на сайте apress.com и создать собственные анимации.

Чтобы освоить систему анимации Angular, необходимо понимать отношения между разными структурными блоками, используемыми для определения и применения анимации:

1. Результат вычисления выражения привязки данных сообщает Angular, какое состояние анимации назначается управляющему элементу.
2. Цель привязки данных сообщает Angular, какая цель анимации определяет стили CSS для состояния элемента.
3. Состояние сообщает Angular, какие стили CSS должны быть применены к элементу.
4. Переход сообщает Angular, как следует применять стили CSS, если вычисление выражения привязки данных приводит к изменению состояния элемента.

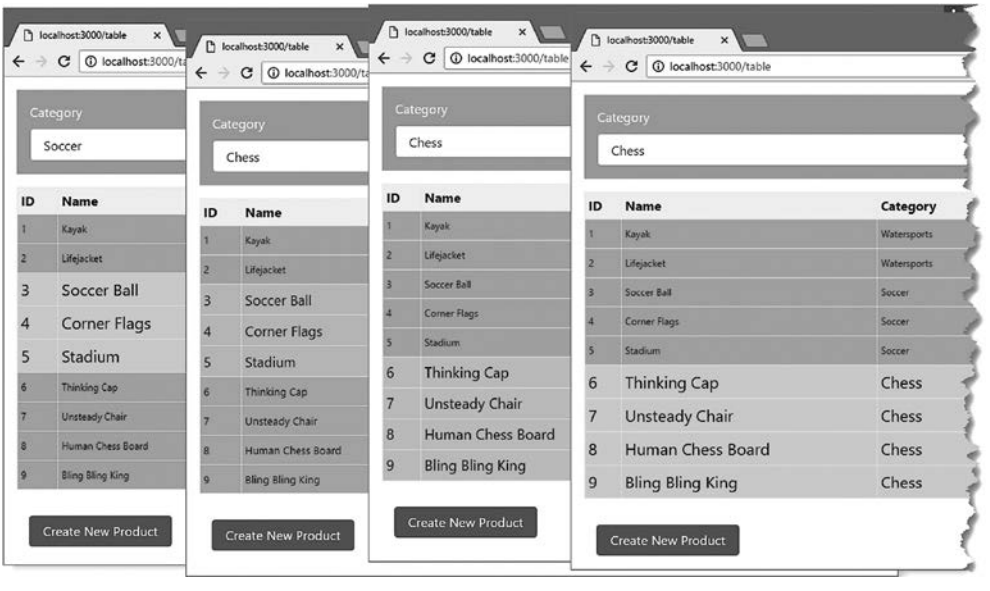


Рис. 28.4. Переходы между состояниями анимации

Помните эти четыре момента, читая оставшуюся часть этой главы, и вам будет намного проще понять систему анимации.

Встроенные состояния анимации

Состояния анимации используются для определения конечного результата анимации: стили, которые должны быть применены к элементу, группируются с именем, которое может быть выбрано триггером анимации. Angular предоставляет два встроенных состояния, которые упрощают управление внешним видом элементов (табл. 28.5).

Таблица 28.5. Встроенные состояния анимации

Состояние	Описание
*	Резервное состояние; применяется, если элемент не находится ни в одном из других состояний, определяемых триггером
void	Элементы находятся в состоянии void, если они не являются частью шаблона. Например, если выражение директивы ngIf дает результат false, управляющий элемент находится в состоянии void. Это состояние используется для анимации добавления и удаления элементов (см. следующий раздел)

Звездочка (*) обозначает специальное состояние, которое должно назначаться всем элементам, не находящимся ни в одном из других состояний, определяемых триггером анимации. В листинге 28.9 резервное состояние добавляется для всех анимаций нашего приложения.

Листинг 28.9. Использование резервного состояния в файле table.animations.ts

```
import { trigger, style, state, transition, animate } from "@angular/core";

export const HighlightTrigger = trigger("rowHighlight", [
    state("selected", style({
        backgroundColor: "lightgreen",
        fontSize: "20px"
    })),
    state("notselected", style({
        backgroundColor: "lightsalmon",
        fontSize: "12px"
    })),
    state("*", style({
        border: "solid black 2px"
    })),
    transition("selected => notselected", animate("200 ms")),
    transition("notselected => selected", animate("400 ms"))
]);
```

В нашем примере после выбора значения в элементе `select` элементам назначается только состояние `selected` или `notselected`. Резервное состояние определяет группу стилей, которая будет назначаться элементам до их перехода в одно из этих состояний (рис. 28.5).

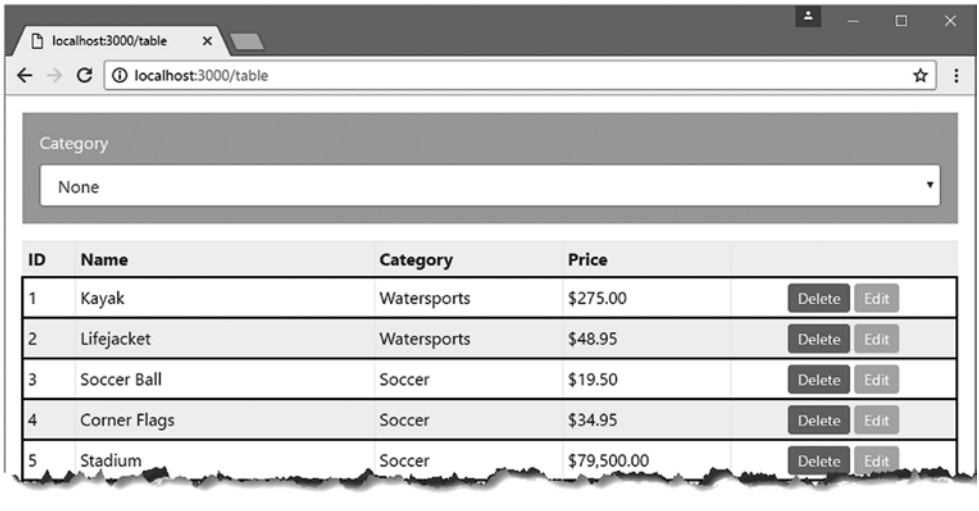


Рис. 28.5. Использование резервного состояния

Переходы элементов

Истинная мощь системы анимации заключена в переходах; именно переходы сообщают Angular, как должны осуществляться изменения состояний. Ниже описаны различные способы создания и использования переходов.

Создание переходов для встроенных состояний

Встроенные состояния из табл. 28.5 могут использоваться в переходах. Для упрощения конфигурации анимации можно воспользоваться резервным состоянием, представляющим любое состояние (листинг 28.10).

Листинг 28.10. Использование резервного состояния в переходе в файле table.animations.ts

```
import { trigger, style, state, transition, animate } from "@angular/core";

export const HighlightTrigger = trigger("rowHighlight", [
    state("selected", style({
        backgroundColor: "lightgreen",
        fontSize: "20px"
    })),
    state("notselected", style({
        backgroundColor: "lightsalmon",
        fontSize: "12px"
    })),
    state("*", style({
        border: "solid black 2px"
    })),
    transition("* => notselected", animate("200 ms")),
    transition("* => selected", animate("400 ms"))
]);
```

Переходы в листинге сообщают Angular, как поступать с изменениями одного состояния в состояния `notselected` и `selected`. В результате Angular анимирует переход, когда пользователь в первый раз выбирает категорию при помощи элемента `select`, а также если в будущем будет выбрана категория `None`.

Анимация добавления и удаления элементов

Состояние `void` используется для определения переходов при добавлении или удалении элементов из шаблона (листинг 28.11).

Листинг 28.11. Использование состояния void в файле table.animations.ts

```
import { trigger, style, state, transition, animate } from "@angular/core";

export const HighlightTrigger = trigger("rowHighlight", [
    state("selected", style({
        backgroundColor: "lightgreen",
        fontSize: "20px"
    })),
    state("notselected", style({
        backgroundColor: "lightsalmon",
        fontSize: "12px"
    })),
    state("void", style({
```

```
        opacity: 0
    })),
    transition("* => notselected", animate("200 ms")),
    transition("* => selected", animate("400 ms")),
    transition("void => *", animate("500 ms"))
]);
```

Листинг включает определение состояния `void`, которое задает свойству `opacity` значение 0, которое делает элемент прозрачным — и, как следствие, невидимым. Также имеется переход, который сообщает Angular, как выполнять анимацию из состояния `void` в любое другое состояние. В результате строки таблицы постепенно «проявляются», когда браузер увеличивает значение `opacity`, до достижения порога (рис. 28.6).

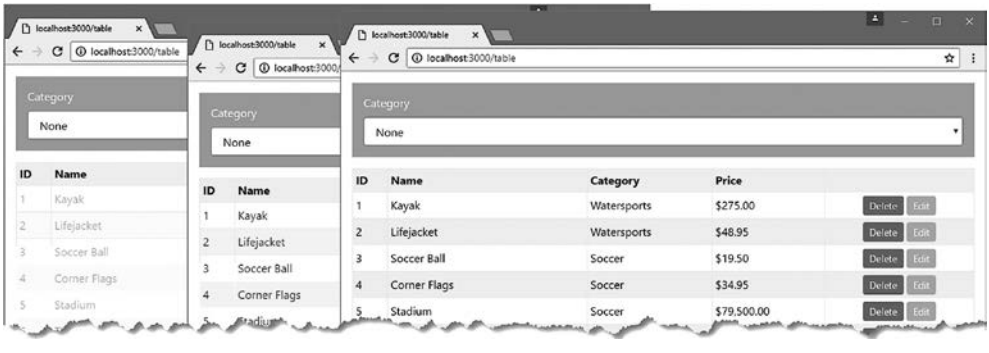


Рис. 28.6. Анимация добавления элемента

ПРИМЕЧАНИЕ

Angular предоставляет псевдонимы для переходов со встроенными состояниями. Вместо void => * можно использовать псевдоним :enter, а вместо * => void — псевдоним :leave.

Управление анимацией переходов

Все примеры, рассматривавшиеся до настоящего момента, использовали функцию `animate` в простейшей форме, в которой задается продолжительность перехода между двумя состояниями:

```
...
transition("void => *", animate("500 ms"))
...
```

Строковый аргумент, передаваемый методу `animate`, также может использоваться для более точного управления анимацией переходов: вы можете определить начальную задержку и указать, сколько должно вычисляться промежуточных значений стилевых свойств.

ВЫРАЖЕНИЕ ПРОДОЛЖИТЕЛЬНОСТИ АНИМАЦИЙ

Продолжительность анимаций выражается в формате временных значений CSS — строк, содержащих одно или несколько чисел с суффиксами s (секунды) или ms (миллисекунды). Например, в следующем примере задается продолжительность 500 миллисекунд:

```
...
transition("void => *", animate("500 ms"))
...
```

Формат выражения продолжительности достаточно гибок; например, то же значение может быть выражено в долях секунды:

```
...
transition("void => *", animate("0.5 s"))
...
```

Я рекомендую выбрать для проекта одну единицу времени (какую именно — неважно), чтобы избежать путаницы.

Определение временной функции

Временная функция отвечает за вычисление промежуточных значений свойств CSS в процессе перехода. Временные функции, определяемые как часть спецификации Web Animations, перечислены в табл. 28.6.

Таблица 28.6. Временные функции анимации

Имя	Описание
linear	Функция изменяет значение с равными приращениями (используется по умолчанию)
ease-in	Функция начинает с малых приращений, которые увеличиваются со временем; в результате анимация начинается медленно и постепенно ускоряется
ease-out	Функция начинает с больших приращений, которые уменьшаются со временем; в результате анимация начинается быстро и постепенно замедляется
ease-in-out	Функция начинает с больших приращений, которые уменьшаются вплоть до средней точки, после которой они снова начинают расти. В результате анимация начинается быстро, замедляется в середине, а потом снова ускоряется
cubic-bezier	Функция используется для создания промежуточных значений по кривой Безье. За подробностями обращайтесь по адресу http://w3c.github.io/web-animations/#time-transformations

В листинге 28.12 временная функция применяется к одному из переходов в примере. Временная функция задается после продолжительности в аргументе функции `animate`.

Листинг 28.12. Применение временной функции в файле table.animations.ts

```
import { trigger, style, state, transition, animate } from "@angular/core";

export const HighlightTrigger = trigger("rowHighlight", [
    state("selected", style({
        backgroundColor: "lightgreen",
        fontSize: "20px"
    })),
    state("notselected", style({
        backgroundColor: "lightsalmon",
        fontSize: "12px"
    })),
    state("void", style({
        opacity: 0
    })),
    transition("* => notselected", animate("200 ms")),
    transition("* => selected", animate("400 ms ease-in")),
    transition("void => *", animate("500 ms"))
]);
```

Определение начальной задержки

Методу `animate` можно передать начальную задержку, которая может использоваться для приостановления анимаций при одновременном выполнении нескольких переходов. Задержка задается вторым значением в аргументе, передаваемом функции `animate` (листинг 28.13).

Листинг 28.13. Добавление начальной задержки в файле table.animations.ts

```
import { trigger, style, state, transition, animate } from "@angular/core";

export const HighlightTrigger = trigger("rowHighlight", [
    state("selected", style({
        backgroundColor: "lightgreen",
        fontSize: "20px"
    })),
    state("notselected", style({
        backgroundColor: "lightsalmon",
        fontSize: "12px"
    })),
    state("void", style({
        opacity: 0
    })),
    transition("* => notselected", animate("200 ms")),
    transition("* => selected", animate("400 ms 200 ms ease-in")),
    transition("void => *", animate("500 ms"))
]);
```

200-миллисекундная задержка в этом примере соответствует продолжительности анимации, используемой при переходе элемента в состояние `notselected`.

В результате при изменении выбранной категории перед изменением элементов `selected` элементы будут возвращаться в состояние `notselected`.

Использование дополнительных стилей во время перехода

Функция `animate` может получать во втором аргументе группу стилей (листинг 28.14). Эти стили шаг за шагом применяются к управляющему элементу на протяжении анимации.

Листинг 28.14. Определение стилей перехода в файле table.animations.ts

```
import { trigger, style, state, transition, animate } from "@angular/core";

export const HighlightTrigger = trigger("rowHighlight", [
    state("selected", style({
        backgroundColor: "lightgreen",
        fontSize: "20px"
    })),
    state("notselected", style({
        backgroundColor: "lightsalmon",
        fontSize: "12px"
    })),
    state("void", style({
        opacity: 0
    })),
    transition("* => notselected", animate("200 ms")),
    transition("* => selected",
        animate("400 ms 200 ms ease-in",
            style({
                backgroundColor: "lightblue",
                fontSize: "25px"
            }))
    ),
    transition("void => *", animate("500 ms"))
]);
```

Это изменение приводит к тому, что при переходе элемента в состояние `selected` его внешний вид изменяется анимацией с переходом к цвету фона `lightblue` с размером шрифта 25 пикселов. В конце анимации применяются стили, определяемые состоянием `selected`.

Резкое изменение внешнего вида в конце анимации выглядит некрасиво. Другое возможное решение — заменить второй аргумент функции `transition` массивом анимаций. Определенные таким образом анимации будут применяться к элементу последовательно, и последняя анимация (при условии, что она не определяет группу стилей) будет использоваться для перехода к стилям, определяемым состоянием. В листинге 28.15 эта возможность используется для добавления двух дополнительных анимаций, последняя из которых применяет стили, определяемые состоянием `selected`.

Листинг 28.15. Использование нескольких анимаций при переходе в файле table.animations.ts

```
import { trigger, style, state, transition, animate } from "@angular/core";

export const HighlightTrigger = trigger("rowHighlight", [
    state("selected", style({
        backgroundColor: "lightgreen",
        fontSize: "20px"
    })),
    state("notselected", style({
        backgroundColor: "lightsalmon",
        fontSize: "12px"
    })),
    state("void", style({
        opacity: 0
    })),
    transition("* => notselected", animate("200 ms")),
    transition("* => selected",
        [animate("400 ms 200 ms ease-in",
            style({
                backgroundColor: "lightblue",
                fontSize: "25px"
            })),
            animate("250 ms", style({
                backgroundColor: "lightcoral",
                fontSize: "30px"
            })),
            animate("200 ms")]
    ),
    transition("void => *", animate("500 ms"))
]);
```

Переход содержит три анимации, последняя из которых применяет стили, определенные состоянием `selected`. Последовательность анимаций описана в табл. 28.7.

Таблица 28.7. Последовательность анимаций при переходе в состояние selected

Продолжительность	Стилевые свойства и их значения
400 миллисекунд	backgroundColor: lightblue; fontSize: 25px
250 миллисекунд	backgroundColor: lightcoral; fontSize: 30px
200 миллисекунд	backgroundColor: lightgreen; fontSize: 20px

Выберите категорию в элементе `select`, чтобы понаблюдать за последовательностью анимаций. На рис. 28.7 показано по одному кадру из каждой анимации.

Параллельные анимации

Angular может выполнять анимации одновременно, с изменением разных свойств CSS за разные периоды времени. Параллельные анимации передаются функции `group` (листинг 28.16).

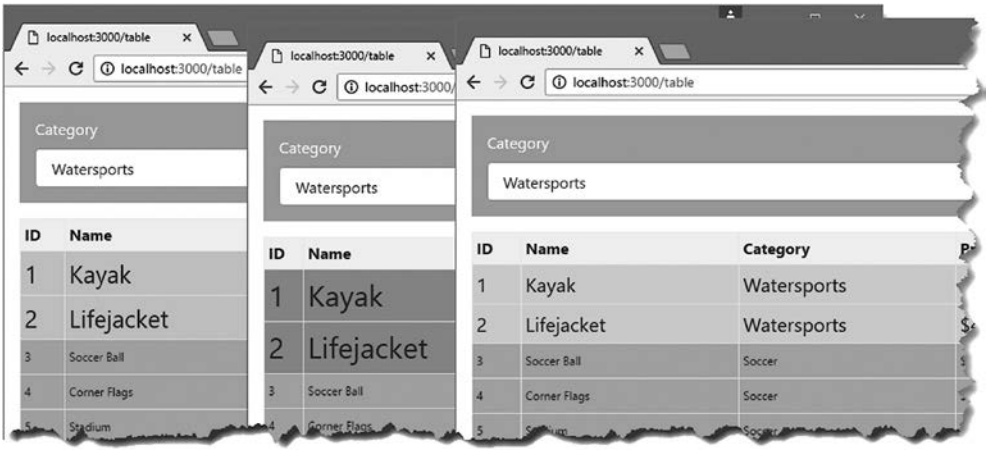


Рис. 28.7. Переход с несколькими анимациями

Листинг 28.16. Выполнение параллельных анимаций в файле table.animations.ts

```
import { trigger, style, state, transition, animate, group } from "@angular/core";

export const HighlightTrigger = trigger("rowHighlight", [
    state("selected", style({
        backgroundColor: "lightgreen",
        fontSize: "20px"
    })),
    state("notselected", style({
        backgroundColor: "lightsalmon",
        fontSize: "12px"
    })),
    state("void", style({
        opacity: 0
    })),
    transition("* => notselected", animate("200 ms")),
    transition("* => selected",
        [animate("400 ms 200 ms ease-in",
            style({
                backgroundColor: "lightblue",
                fontSize: "25px"
            })),
            group([
                animate("250 ms", style({
                    backgroundColor: "lightcoral",
                })),
                animate("450 ms", style({
                    fontSize: "30px"
                })),
            ]),
            animate("200 ms")]
    ),
    transition("void => *", animate("500 ms"))
]);
```

В листинге одна из анимаций заменяется парой параллельных анимаций. Анимации свойств `backgroundColor` и `fontSize` начинаются одновременно, но имеют разную продолжительность. Когда обе анимации группы завершатся, Angular перейдет к последней анимации, которая использует стили, определенные в состоянии.

Группы стилей анимации

В результате анимации Angular элемент переводится в новое состояние и к нему применяется стилевое оформление с использованием свойств и значений соответствующей группы стилей. В этом разделе рассматриваются некоторые способы использования групп стилей.

ПРИМЕЧАНИЕ

Не все свойства CSS могут использоваться в анимации. Впрочем, даже те, которые могут, в одних браузерах поддерживаются лучше, чем в других. Как правило, лучшие результаты достигаются со свойствами, значения которых легко интерполируются; это позволяет браузеру сгенерировать плавный переход между состояниями элементов. Это означает, что в анимациях хорошо работают свойства, значения которых являются цветами или числовыми значениями, — цвета фона и шрифта, прозрачность, размеры элементов и обрамление. Полный список свойств, которые могут использоваться в системе анимации, доступен по адресу https://www.w3.org/TR/css3-transitions/#animatable-properties.

Определение общих стилей в группах для многократного использования

Когда вы начнете создавать более сложные анимации и применять их в своих приложениях, со временем неизбежно выяснится, что некоторые значения свойств CSS приходится применять в разных местах. Функция `style` может получать массив объектов, которые объединяются для создания единого набора стилей в группе. Это означает, что вы можете избавиться от дублирования, определяя объекты, содержащие общие стили, и использовать их в разных группах стилей (листинг 28.17). (Для упрощения примера я также удалил серию стилей, определенную в предыдущем разделе.)

Листинг 28.17. Определение общих стилей в файле table.animations.ts

```
import { trigger, style, state, transition, animate, group } from "@angular/core";
const commonStyles = {
    border: "black solid 4px",
    color: "white"
};

export const HighlightTrigger = trigger("rowHighlight", [
    state("selected", style([commonStyles, {
        backgroundColor: "lightgreen",
        fontSize: "20px"
    }])),
    state("notselected", style([commonStyles, {
```

```
        backgroundColor: "lightsalmon",
        fontSize: "12px",
        color: "black"
    }])),
    state("void", style({
        opacity: 0
    })),
    transition("* => notselected", animate("200 ms")),
    transition("* => selected", animate("400 ms 200 ms ease-in")),
    transition("void => *", animate("500 ms"))
]);
```

Объект `commonStyles` определяет значения для свойств `border` и `color` и передается функции `style` в массиве вместе с обычными объектами стилей. Angular последовательно обрабатывает объекты стилей; это означает, что стилевые значения могут переопределяться в более поздних объектах. Например, второй объект стиля для состояния `notselected` переопределяет общее значение свойства `color` специализированным значением. В результате стили обоих состояний анимации включают общее значение свойства `border`, а стили состояния `selected` также используют общее значение свойства `color` (рис. 28.8).

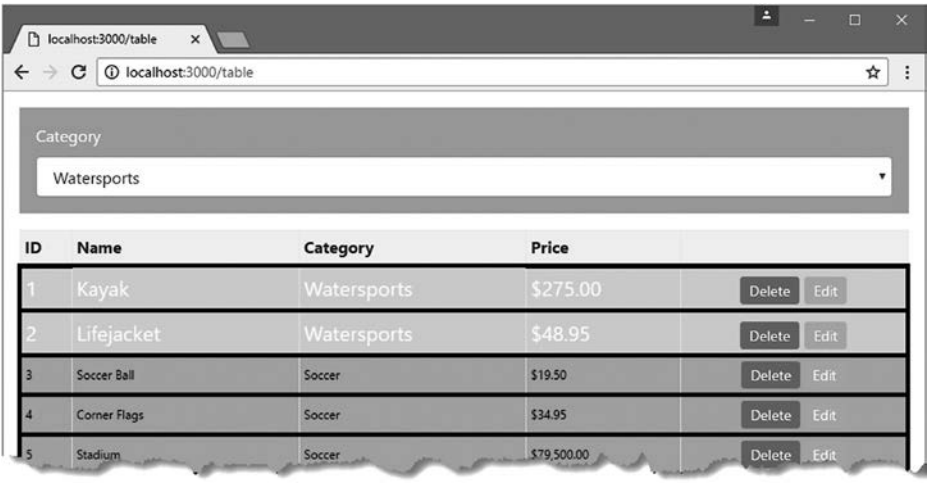


Рис. 28.8. Определение общих свойств

Использование преобразований элементов

Во всех примерах, рассматривавшихся ранее в этой главе, использовались анимированные свойства, влиявшие на отдельные аспекты внешнего вида элемента: цвет фона, размер шрифта, прозрачность и т. д. Анимации также могут использоваться для применения эффектов преобразования элементов CSS, выполняющих перемещение, изменение размеров, повороты или деформацию элементов. Для применения этих эффектов в группе стилей определяется свойство `transform` (листинг 28.18).

Листинг 28.18. Использование преобразования элемента в файле table.animations.ts

```
import { trigger, style, state, transition, animate, group } from "@angular/core";

const commonStyles = {
    border: "black solid 4px",
    color: "white"
};

export const HighlightTrigger = trigger("rowHighlight", [
    state("selected", style([commonStyles, {
        backgroundColor: "lightgreen",
        fontSize: "20px"
    }])),
    state("notselected", style([commonStyles, {
        backgroundColor: "lightsalmon",
        fontSize: "12px",
        color: "black"
    }])),
    state("void", style({
        transform: "translateX(-50%)"
    })),
    transition("* => notselected", animate("200 ms")),
    transition("* => selected", animate("400 ms 200 ms ease-in")),
    transition("void => *",  animate("500 ms"))
]);
```

Свойству `transform` присваивается значение `translateX(50%)`, которое приказывает Angular переместить элемент на 50% его длины по оси *x*. Свойство `transform` применяется к состоянию `void`; это означает, что оно будет использоваться с элементами, добавляемыми в шаблон. Анимация содержит переход из состояния `void` в любое другое состояние и приказывает Angular выполнить анимацию изменений за 500 миллисекунд.

В результате новые элементы сначала сдвигаются влево, а затем плавно скользят в стандартную позицию за половину секунды (рис. 28.9).

В табл. 28.8 перечислены преобразования, которые могут применяться к элементам.

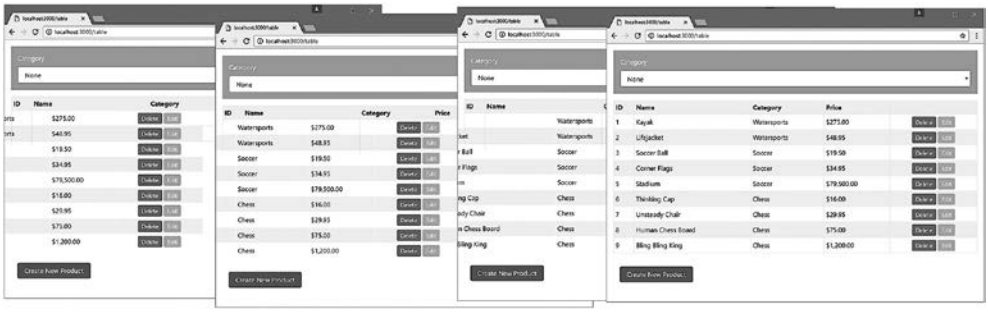


Рис. 28.9. Преобразование элемента

ПРИМЕЧАНИЕ

В одном свойстве transform можно применить сразу несколько преобразований. Для этого разделите их пробелами, как в следующем примере: `transform: "scale(1.1, 1.1) rotate(10 deg)"`.

Таблица 28.8. Функции преобразования CSS

Функция	Описание
translateX(offset)	Функция перемещает элемент по оси x. Величина перемещения может задаваться в процентах или в виде величины смещения (выраженной в пикселах или других единицах длины CSS). С положительными значениями элемент перемещается вправо, а с отрицательными — влево
translateY(offset)	Функция перемещает элемент по оси y
translate(xOffset, yOffset)	Функция перемещает элемент по обеим осям
scaleX(amount)	Функция масштабирует элемент по оси x. Коэффициент масштабирования выражается в процентах от обычного размера элемента (например, со значением 0,5 элемент уменьшается до 50 % от исходной ширины, а со значением 2,0 ширина удваивается)
scaleY(amount)	Функция масштабирует элемент по оси y
scale(xAmount, yAmount)	Функция масштабирует элемент по обеим осям
rotate(angle)	Функция поворачивает элемент по часовой стрелке. Величина поворота задается размером угла (например, 90 deg или 3,14 rad)
skewX(angle)	Функция выполняет деформацию сдвига элемента по оси x на заданный угол, который выражается так же, как для функции rotate
skewY(angle)	Функция выполняет деформацию сдвига элемента по оси y на заданный угол, который выражается так же, как для функции rotate
skew(xAngle, yAngle)	Функция выполняет деформацию сдвига элемента по обеим осям

Применение стилей фреймворка CSS

Если вы используете фреймворк CSS (например, Bootstrap), возможно, вы предпочтете назначать элементам классы вместо того, чтобы определять группы свойств. Встроенной поддержки для непосредственной работы с классами CSS не существует, но при помощи API DOM и CSSOM (CSS Object Model) вы сможете просмотреть загруженные таблицы стилей CSS и проверить, применяются ли они к элементу HTML. Чтобы получить набор стилей, определенных классами, создайте файл animationUtils.ts в папке exampleApp/app/core и добавьте код из листинга 28.19.

ВНИМАНИЕ

В приложениях с большим количеством сложных таблиц стилей это решение может потребовать значительных вычислительных ресурсов. Возможно, вам придется внести изменения в код, чтобы он работал с разными браузерами и разными фреймворками CSS.

Листинг 28.19. Содержимое файла animationUtils.ts в папке exampleApp/app/core

```
export function getStylesFromClasses(names: string | string[],
        elementType: string = "div") : { [key: string]: string | number } {

    let elem = document.createElement(elementType);
    (typeof names == "string" ? [names] : names).forEach(c => elem.classList.add(c));

    let result = {};
    for (let i = 0; i < document.styleSheets.length; i++) {
        let sheet = document.styleSheets[i] as CSSStyleSheet;
        let rules = sheet.rules || sheet.cssRules;
        for (let j = 0; j < rules.length; j++) {
            if (rules[j].type == CSSRule.STYLE_RULE) {
                let styleRule = rules[j] as CSSStyleRule;
                if (elem.matches(styleRule.selectorText)) {
                    for (let k = 0; k < styleRule.style.length; k++) {
                        result[styleRule.style[k]] =
                            styleRule.style[styleRule.style[k]];
                    }
                }
            }
        }
    }
    return result;
}
```

Метод `getStylesFromClass` получает имя класса (или массив имен классов) и тип элемента, к которому они должны применяться (по умолчанию элемент `div`). Элемент создается, ему назначаются классы, после чего элемент проверяется, чтобы узнать, какие из правил CSS, определенных в таблицах стилей CSS, применимы к нему. Стилевые свойства каждого подходящего стиля добавляются в объект, который затем может использоваться для создания групп стилей анимации Angular (листинг 28.20).

Листинг 28.20. Использование классов Bootstrap в файле table.animations.ts

```
import { trigger, style, state, transition, animate, group } from "@angular/core";
import { getStylesFromClasses } from "./animationUtils";

export const HighlightTrigger = trigger("rowHighlight", [
    state("selected", style(getStylesFromClasses(["bg-success", "h2"]))),
    state("notselected", style(getStylesFromClasses("bg-info"))),
    state("void", style({
```

```
        transform: "translateX(-50%)"
    })),
    transition("* => notselected", animate("200 ms")),
    transition("* => selected", animate("400 ms 200 ms ease-in")),
    transition("void => *", animate("500 ms"))
]);
```

Состояние `selected` использует стили, определенные в классах Bootstrap `bg-success` и `h2`, а состояние `notselected` использует стили, определяемые классом Bootstrap `bg-info`. Результат показан на рис. 28.10.

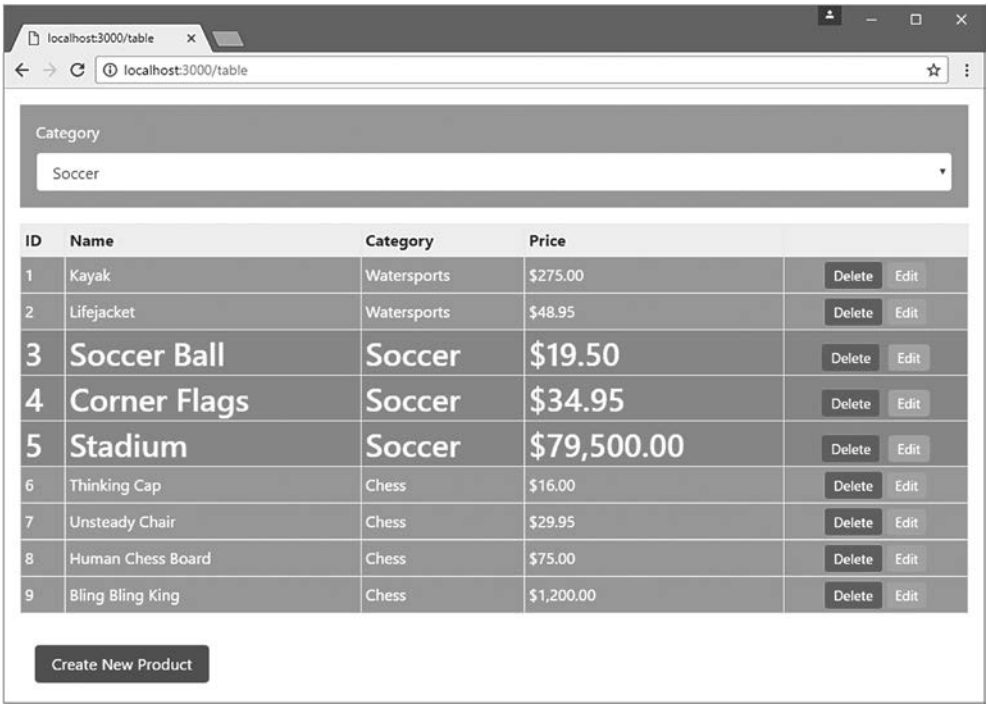


Рис. 28.10. Использование стилей фреймворка CSS в анимациях Angular

События триггеров анимации

По сравнению с другими структурными блоками анимации триггеры относительно просты. Они создают обертку для наборов состояний и переходов и формируют связь между элементами, к которым применяются анимации.

Мы не рассмотрели еще одну возможность: директива, которая применяет триггеры к элементам, генерирует события, относящиеся к выполняемой анимации. Эти события представляются классом `AnimationTransitionEvent`; свойства, определяемые этим классом, перечислены в табл. 28.9.

Таблица 28.9. Свойства AnimationTransitionEvent

Имя	Описание
fromState	Свойство возвращает состояние, из которого выходит элемент
toState	Свойство возвращает состояние, в которое входит элемент
totalTime	Свойство возвращает продолжительность анимации

Директива анимации предоставляет свойства `start` и `done`, которые могут использоваться в привязках событий (листинг 28.21).

Листинг 28.21. Привязка событий анимации в файле table.component.html

```
<div class="form-group bg-info p-a-1">
    <label>Category</label>
    <select [(ngModel)]="highlightCategory" class="form-control">
        <option value="">None</option>
        <option *ngFor="let category of categories">
            {{category}}
        </option>
    </select>
</div>
<table class="table table-sm table-bordered table-striped">
    <tr>
        <th>ID</th><th>Name</th><th>Category</th><th>Price</th><th></th>
    </tr>
    <tr *ngFor="let item of getProducts()"
            [@rowHighlight]="getRowState(item.category)"
            (@rowHighlight.start)="writeAnimationEvent($event, item.name, true)"
            (@rowHighlight.done)="writeAnimationEvent($event, item.name, false)">
        <td style="vertical-align:middle">{{item.id}}</td>
        <td style="vertical-align:middle">{{item.name}}</td>
        <td style="vertical-align:middle">{{item.category}}</td>
        <td style="vertical-align:middle">
            {{item.price | currency:"USD":true }}
        </td>
        <td class="text-xs-center">
            <button class="btn btn-danger btn-sm"
                    (click)="deleteProduct(item.id)">
                Delete
            </button>
            <button class="btn btn-warning btn-sm"
                [routerLink]="['/form', 'edit', item.id]">
                Edit
            </button>
        </td>
    </tr>
</table>
<div class="col-xs-12 p-t-1">
    <button class="btn btn-primary" routerLink="/form/create">
        Create New Product
    </button>
</div>
```

Целью этих привязок является метод компонента с именем `writeAnimationEvent`, которому передается событие, название товара, выводимого элементом `tr` и признак того, является ли это событие стартовым. В листинге 28.22 приведена реализация метода в компоненте, которая просто выводит информацию на консоль JavaScript в браузере.

Листинг 28.22. Обработка событий анимации в файле table.component.ts

```
import { Component, Inject } from "@angular/core";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
import { ActivatedRoute } from "@angular/router";
import { HighlightTrigger } from "./table.animations";
import { AnimationTransitionEvent } from "@angular/core";

@Component({
    selector: "paTable",
    moduleId: module.id,
    templateUrl: "table.component.html",
    animations: [HighlightTrigger]
})
export class TableComponent {
    category: string = null;

    constructor(private model: Model, activeRoute: ActivatedRoute) {
        activeRoute.params.subscribe(params => {
            this.category = params["category"] || null;
        })
    }

    getProduct(key: number): Product {
        return this.model.getProduct(key);
    }

    getProducts(): Product[] {
        return this.model.getProducts()
            .filter(p => this.category == null || p.category == this.category);
    }

    get categories(): string[] {
        return this.model.getProducts()
            .map(p => p.category)
            .filter((category, index, array) => array.indexOf(category) == index);
    }

   deleteProduct(key: number) {
       this.model.deleteProduct(key);
   }

   highlightCategory: string = "";

   getRowState(category: string): string {
       return this.highlightCategory == "" ? "" :
```

```
            this.highlightCategory == category ? "selected" : "notselected";
    }

    writeAnimationEvent(event: AnimationTransitionEvent,
        name: string, start: boolean) {
        console.log("Animation " + name + " " + (start ? 'Start' : 'Done')
            + " from: " + event.fromState + " to: " + event.toState + " time: "
            + event.totalTime);
    }
}
```

При выборе категории на консоли JavaScript выводится информация обо всех анимациях строки таблицы:

```
Animation Lifejacket Start from: notselected to: selected time: 600
Animation Soccer Ball Start from: selected to: notselected time: 200
Animation Corner Flags Start from: selected to: notselected time: 200
Animation Stadium Start from: selected to: notselected time: 200
Animation Soccer Ball Done from: selected to: notselected time: 200
Animation Corner Flags Done from: selected to: notselected time: 200
Animation Stadium Done from: selected to: notselected time: 200
Animation Kayak Done from: notselected to: selected time: 600
Animation Lifejacket Done from: notselected to: selected time: 600
```

Итоги

В этой главе вы узнали о системе анимации Angular и о том, как она использует привязки данных для анимации изменений в состоянии приложения. Следующая глава посвящена функциональности Angular, предназначенной для поддержки модульного тестирования.

29 Модульное тестирование в Angular

В этой главе описаны средства Angular для модульного тестирования компонентов и директив. Некоторые структурные блоки Angular, такие как каналы и службы, без особых проблем тестируются в изоляции от других при помощи базовых инструментов тестирования, созданных в начале главы. Компоненты (и в меньшей степени директивы) связаны сложными взаимодействиями со своими управляющими элементами и контентом их шаблонов, поэтому для них потребуются специальные средства. В табл. 29.1 средства модульного тестирования Angular представлены в контексте.

Таблица 29.1. Средства модульного тестирования Angular в контексте

Вопрос	Ответ
Что это такое?	Компонентам и директивам Angular требуются специальные средства тестирования, чтобы их взаимодействия с другими частями инфраструктуры приложения можно было изолировать для анализа
Для чего они нужны?	Изолированные модульные тесты могут обращаться к базовой логике, предоставляемой классом, реализующим компонент или директиву, но не отслеживают взаимодействия с управляющими элементами, службами, шаблонами и другими важными средствами Angular
Как они используются?	Angular предоставляет тестовую среду, которая позволяет создать реалистичное окружение и использовать его для выполнения модульных тестов
Есть ли у них недостатки или скрытые проблемы?	Как и многие аспекты Angular, средства модульного тестирования сложны. Возможно, вы не сразу научитесь легко писать и запускать модульные тесты, а также изолировать нужные части приложения для тестирования
Есть ли альтернативы?	Как упоминалось ранее, проводить модульное тестирование ваших проектов не обязательно. Но если вы захотите выполнить модульное тестирование, вам понадобятся средства Angular, описанные в этой главе

В табл. 29.2 приведена краткая сводка материала главы.

Таблица 29.2. Сводка материала главы

Проблема	Решение	Листинг
Подготовка к модульному тестированию	Добавьте в проект и настройте пакеты Karma и Jasmine	1–8
Выполнение простейшего теста с компонентом	Инициализируйте тестовый модуль и создайте экземпляр компонента. Если компонент использует внешний шаблон, необходимо выполнить дополнительный шаг компиляции	9–12, 14–16
Тестирование привязок данных компонента	Используйте класс DebugElement для получения информации о шаблоне	13
Тестирование реакции компонента на события	Инициируйте события элементом при помощи DebugElement	14–19
Тестирование выходных свойств компонента	Подпишитесь на объект EventEmitter, созданный компонентом	20, 21
Тестирование входных свойств компонента	Создайте тестовый компонент, шаблон которого применяет тестируемый компонент	22, 23
Выполнение теста, зависящего от асинхронной операции	Используйте метод whenStable, чтобы отложить тест до обработки эффекта операции	24, 25
Тестирование директивы	Создайте тестовый компонент, шаблон которого применяет тестируемую директиву	26, 27

ПРОВОДИТЬ ЛИ МОДУЛЬНОЕ ТЕСТИРОВАНИЕ?

По поводу модульного тестирования существуют разные мнения. Эта глава предполагает, что вы хотите провести модульное тестирование; она показывает, как настроить средства тестирования и применить их к компонентам и директивам Angular. Глава не содержит вводной информации по теме модульного тестирования, и я не пытаюсь убедить скептически настроенных читателей, что модульное тестирование — дело стоящее. Если вам понадобится введение в модульное тестирование, хорошая статья доступна по адресу https://en.wikipedia.org/wiki/Unit_testing.

Мне нравится модульное тестирование, и я использую его в своих проектах — но не во всех и не настолько часто, как можно было бы ожидать. Я стараюсь писать модульные тесты для тех функций и аспектов, которые с большой вероятностью создадут проблемы и которые могут стать источником ошибок при развертывании. В таких ситуациях модульное тестирование помогает правильно выстроить мои представления о том, как лучше реализовать то, что мне понадобится. Даже когда я просто думаю о том, что нужно для тестирования, я начинаю представлять себе потенциальные проблемы, причем еще до того, как начну иметь дело с реальными ошибками и дефектами.

Вместе с тем модульное тестирование — инструмент, а не религия, и только вы знаете, какой объем тестирования вам нужен. Если вы сочтете, что модульное тестирование в вашем приложении бесполезно или у вас есть другая методология, которая вам подходит лучше, не думайте, что модульное тестирование обязательно использовать просто потому, что оно в моде. (С другой стороны, если другой методологии нет и вы вообще не проводите тестирование, скорее всего, ваши ошибки будут находить пользователи, а это обычно нежелательно.)

Подготовка проекта

В этой главе мы продолжим использовать проект exampleApp из предыдущих глав. Для проведения модульного тестирования лучше убрать все лишнее, поэтому в листинге 29.1 конфигурация маршрутизации изменяется так, чтобы функциональный модуль ondemand загружался по умолчанию.

Листинг 29.1. Изменение конфигурации маршрутизации в файле app.routing.ts

```
import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "./core/table.component";
import { FormComponent } from "./core/form.component";
import { NotFoundComponent } from "./core/notFound.component";
import { ProductCountComponent } from "./core/productCount.component";
import { CategoryCountComponent } from "./core/categoryCount.component";
import { ModelResolver } from "./model/model.resolver";
import { TermsGuard } from "./terms.guard";
import { UnsavedGuard } from "./core/unsaved.guard";
import { LoadGuard } from "./load.guard";

const routes: Routes = [
    {
        path: "ondemand",
        loadChildren: "app/ondemand/ondemand.module#OndemandModule"
    },
    { path: "", redirectTo: "/ondemand", pathMatch: "full" }
]

export const routing = RouterModule.forRoot(routes);
```

Модуль содержит простые компоненты, которые будут использоваться для демонстрации возможностей модульного тестирования. Для простоты в листинге 29.2 поправляется шаблон, отображаемый компонентом верхнего уровня в функциональном модуле.

Листинг 29.2. Упрощение шаблона в файле ondemand.component.html

```
<div class="col-xs-12 p-a-1">
    <router-outlet></router-outlet>
</div>
<div class="col-xs-6">
    <router-outlet name="left"></router-outlet>
</div>
<div class="col-xs-6">
    <router-outlet name="right"></router-outlet>
</div>
<div class="p-a-1">
    <button class="btn btn-secondary m-t-1" routerLink="/ondemand">Normal</button>
    <button class="btn btn-secondary m-t-1" routerLink="/ondemand/swap">Swap</
button>
</div>
```

Добавление пакетов тестирования

Чтобы настроить и выполнить модульные тесты, следует добавить в проект несколько пакетов (листинг 29.3).

ВНИМАНИЕ

Для запуска приложений Angular можно использовать любой современный браузер, но примеры этой главы предполагают, что для запуска модульных тестов используется Google Chrome. Установите Chrome, если вы хотите воспроизвести наши примеры.

Листинг 29.3. Добавление пакетов модульного тестирования в файл package.json

```
{
  "dependencies": {
    "@angular/common": "2.2.0",
    "@angular/compiler": "2.2.0",
    "@angular/core": "2.2.0",
    "@angular/platform-browser": "2.2.0",
    "@angular/platform-browser-dynamic": "2.2.0",
    "@angular/upgrade": "2.2.0",
    "@angular/forms": "2.2.0",
    "@angular/http": "2.2.0",
    "@angular/router": "3.2.0",
    "reflect-metadata": "0.1.8",
    "rxjs": "5.0.0-beta.12",
    "zone.js": "0.6.26",
    "core-js": "2.4.1",
    "classlist.js": "1.1.20150312",
    "systemjs": "0.19.40",
    "bootstrap": "4.0.0-alpha.4",
    "intl": "1.2.4",
    "html5-history-api": "4.2.7",
    "web-animations-js": "2.2.2"
  },

  "devDependencies": {
    "lite-server": "2.2.2",
    "typescript": "2.0.2",
    "typings": "1.3.2",
    "concurrently": "2.2.0",
    "systemjs-builder": "0.15.32",
    "json-server": "0.8.21",
    "jasmine-core": "2.5.2",
    "karma": "1.3.0",
    "karma-jasmine": "1.0.2",
    "karma-chrome-launcher": "2.0.0"
  },

  "scripts": {
    "start": "concurrently \"npm run tscwatch\" \"npm run lite\" \"npm run json\"
",
    "tsc": "tsc",
```

```
    "tscwatch": "tsc -w",
    "lite": "lite-server",
    "json": "json-server --p 3500 restData.js",
    "karma": "karma start karma.conf.js",
    "tests": "npm run karma"
    "typings": "typings",
    "postinstall": "typings install"
  }
}
```

Для выполнения модульного тестирования в этой главе используются два разных инструмента. Jasmin — популярный фреймворк модульного тестирования с возможностью определения тестов и проверки их результатов. Код содержащихся в файлах модульных тестов записывается с использованием API Jasmine (он будет описан в разделе «Работа с Jasmine»).

Karma — система исполнения тестов; она отслеживает файлы проекта и запускает тесты, определенные с использованием Jasmine, при обнаружении изменений. Комбинация этих двух инструментов позволяет легко определять и запускать модульные тесты в проектах JavaScript. Тем не менее Angular вносит в этот процесс некоторые нюансы; это означает, что процесс настройки Karma и написания модульных тестов для приложений Angular требует некоторых мер, необязательных для модульного тестирования других разновидностей кода JavaScript.

Сохраните изменения в файле package.json и выполните следующую команду из папки exampleApp, чтобы загрузить и установить новые пакеты:

```
npm install
```

Настройка Karma

Самая сложная часть настройки модульного тестирования — правильная настройка конфигурации. Вероятно, задача потребует ряда проб и ошибок. Проблема в том, что для тестирования приложений Angular необходимо использовать загрузчик модулей JavaScript для обработки зависимостей в приложении, чтобы модульный тест смог обратиться к функциональности Angular, а для этого нужны пакеты JavaScript, от которых зависит Angular. Все начинается с создания файла конфигурации Karma; для этого добавьте файл karma.conf.js в папку exampleApp и добавьте код конфигурации из листинга 29.4.

Листинг 29.4. Содержимое файла karma.conf.js в папке exampleApp

```
module.exports = (config) => { config.set({
frameworks: ["jasmine"],
plugins: [require("karma-jasmine"), require("karma-chrome-launcher")],

files: [
    "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/proxy.js",
```

```
    "node_modules/zone.js/dist/sync-test.js",
    "node_modules/zone.js/dist/jasmine-patch.js",
    "node_modules/zone.js/dist/async-test.js",
    "node_modules/zone.js/dist/fake-async-test.js",
    { pattern: "node_modules/rxjs/**/*.js", included: false, watched: false },
    { pattern: "node_modules/@angular/**/*.js", included: false, watched: false },
    { pattern: "app/**/*.js", included: false, watched: true },
    { pattern: "app/**/*.html", included: false, watched: true },
    { pattern: "app/**/*.css", included: false, watched: true },
    { pattern: "tests/**/*.js", included: false, watched: true },

    { pattern: "karma-test-shim.js", included: true, watched: true },
],
reporters: ["progress"],
port: 9876,
colors: true,
logLevel: config.LOG_INFO,
autoWatch: true,
browsers: ["Chrome"],
singleRun: false
})}
```

Самая важная часть конфигурации — секция `files` — используется для определения конфигурации файлов, задействованных в модульном тестировании. Первая группа записей включает файлы, необходимые для поддержки модульного тестирования в Angular; из нее Karma получает информацию о файлах фреймворка Angular и Reactive Extensions. Вторая группа файлов сообщает Karma, где искать файлы JavaScript, HTML и CSS, а также файлы тестов.

ПРИМЕЧАНИЕ

Я предпочитаю хранить свои модульные тесты отдельно от остального кода приложения в папке с именем tests. Другие разработчики хранят тесты вместе с кодом; в модульном тестировании Angular возможны оба варианта.

Последняя запись определяет файл с именем karma-test-shim.js, отвечающий за настройку тестовой среды, необходимой для Angular. Создайте этот файл в папке exampleApp и добавьте в него код из листинга 29.5.

Листинг 29.5. Содержимое файла karma-test-shim.js в папке exampleApp

```
__karma__.loaded = function () {};

let map = { "rxjs": "node_modules/rxjs" };

var angularModules = ["common", "compiler", "core", "platform-browser",
    "platform-browser-dynamic", "forms", "http", "router"];

angularModules.forEach(module => {
    map[`@angular/${module}`] =
        `node_modules/@angular/${module}/bundles/${module}.umd.js`;
    map[`@angular/${module}/testing`] =
        `node_modules/@angular/${module}/bundles/${module}-testing.umd.js`
```

```
});

System.config({ baseURL: "/base", map: map, defaultJSExtensions: true });

Promise.all([
  System.import("@angular/core/testing"),
  System.import("@angular/platform-browser-dynamic/testing")
]).then(providers => {
    var testing = providers[0];
    var testingBrowser = providers[1];
    testing.TestBed.initTestEnvironment(testingBrowser.BrowserDynamicTestingModule,
      testingBrowser.platformBrowserDynamicTesting());

}).then(() => Promise.all(Object.keys(window.__karma__.files)
    .filter(name => name.endsWith("spec.js"))
    .map(moduleName => System.import(moduleName))))
.then(__karma__.start, __karma__.error);
```

Первая часть файла создает конфигурацию SystemJS, которая будет использоваться для разрешения зависимостей модулей JavaScript. Вторая часть создает объект `Promise` для загрузки базовых модулей тестирования, настраивает инфраструктуру тестирования Angular и импортирует модульные тесты.

ПРИМЕЧАНИЕ

В этой главе я определяю модульные тесты в файлах, имена которых завершаются суффиксом spec.js. В листинге 29.5 эта схема имен используется для нахождения модульных тестов в проекте. Измените эту функцию, если вы используете другую схему назначения имен.

Настройка TypeScript

Чтобы компилятор TypeScript понимал API Jasmine, ему понадобятся аннотации типа. Выполните следующую команду из папки exampleApp, чтобы загрузить и установить информацию типов Jasmine:

```
npm run typings -- install dt~jasmine --save --global
```

Создание простого модульного теста

Чтобы проверить правильность работы модульного тестирования, полезно создать простейший модульный тест. Создайте файл frameworkTest.spec.ts в папке tests и добавьте код из листинга 29.6.

Листинг 29.6. Содержимое файла frameworkTest.spec.ts в папке exampleApp/tests

```
describe("Jasmine Test Environment", () => {
    it("is working", () => expect(true).toBe(true));
});
```

Вскоре мы рассмотрим основы работы с API Jasmine, а на синтаксис пока можно не обращать внимания.

Запуск инструментария

Модульные тесты будут запускаться параллельно с приложением; это позволит нам расширять функциональность приложения и тестов одновременно. Для этого нам потребуются два окна командной строки. В первом окне перейдите в папку exampleApp и выполните следующую команду:

```
npm start
```

Команда запускает компилятор TypeScript, сервер HTTP для разработки и REST-совместимую веб-службу. Открывается новое окно (или вкладка) браузера с контентом, показанным на рис. 29.1.

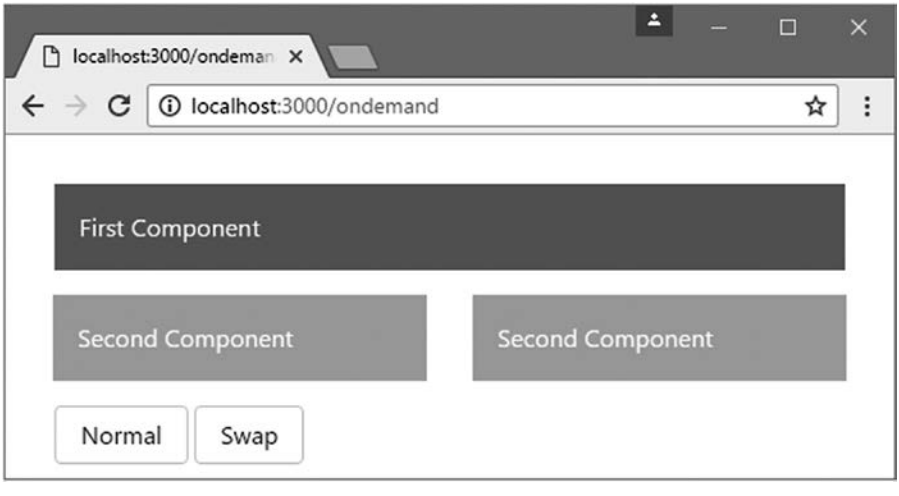


Рис. 29.1. Запуск приложения

Во втором окне командной строки перейдите в папку exampleApp и выполните следующую команду:

```
npm run tests
```

Команда использует запись `scripts`, добавленную в файл package.json в начале главы, и запускает систему исполнения тестов Karma. Открывается новая вкладка (рис. 29.2).

Окно браузера используется для выполнения тестов, но в окне командной строки выводится важная информация:

```
Chrome 54.0.2840 (Windows 10 0.0.0): Executed 1 of 1 SUCCESS (0.002 secs / 0 secs)
```

Из этого сообщения видно, что в проекте был найден один модульный тест, который был успешно выполнен. Каждый раз, когда вы вносите изменение в один из файлов JavaScript в проекте, система находит и исполняет модульные тесты и информация обо всех проблемах выводится в командную строку. Чтобы показать, как выглядит ошибка, в листинге 29.7 модульный тест меняется так, чтобы он не проходил.

Рис. 29.2. Запуск системы исполнения тестов Karma

Листинг 29.7. Включение непрохождения теста в файле frameworkTest.spec.ts

```
describe("Jasmine Test Environment", () => {
    it("is working", () => expect(true).toBe(false));
});
```

Тест не проходит, а на консоль выводится следующее сообщение с описанием обнаруженных проблем:

```
Chrome 54.0.2840 (Windows 10 0.0.0) Jasmine Test Environment is working FAILED
[1]     Expected true to be false.
[1]         at Object.eval (tests/frameworkTest.spec.js:2:56)
[1]         at ZoneDelegate.invoke (node_modules/zone.js/dist/zone.js:232:26)
[1]         at ProxyZoneSpec.onInvoke (node_modules/zone.js/dist/proxy.js:79:39)
[1]         at ZoneDelegate.invoke (node_modules/zone.js/dist/zone.js:231:32)
Chrome 54.0.2840 (Windows 10 0.0.0): Executed 1 of 1 (1 FAILED) ERROR
```

Работа с Jasmine

API Jasmine объединяет методы JavaScript для определения модульных тестов. Полная документация Jasmine доступна по адресу *http://jasmine.github.io*, но в табл. 29.3 описаны самые полезные функции тестирования в Angular.

Таблица 29.3. Полезные методы Jasmine

Имя	Описание
describe(description, function)	Метод используется для группировки взаимосвязанных тестов
beforeEach(function)	Метод используется для назначения задачи, которая должна выполняться перед каждым модульным тестом
afterEach(function)	Метод используется для назначения задачи, которая должна выполняться после каждого модульного теста
it(description, function)	Метод используется для выполнения тестового действия
expect(value)	Метод используется для идентификации результата теста
toBe(value)	Метод задает ожидаемое значение теста

Методы из табл. 29.3 использованы для создания модульного теста в листинге 29.7.

```
...
describe("Jasmine Test Environment", () => {
    it("is working", () => expect(true).toBe(false));
});
...
```

Вы также видите, почему тест не проходит: методы `expect` и `toBe` используются для проверки равенства `true` и `false`. Такого быть не может, поэтому тест не проходит.

Метод `toBe` не единственный способ проверки результата модульного теста. В табл. 29.4 представлены другие методы, предоставляемые Angular.

Таблица 29.4. Полезные методы проверки результатов тестов в Jasmine

Имя	Описание
toBe(value)	Метод проверяет, что результат равен заданному значению (при этом он не обязан быть тем же объектом)
toEqual(object)	Метод проверяет, что результат является тем же объектом, что и заданное значение
toMatch(regexp)	Метод проверяет, что результат соответствует заданному регулярному выражению
toBeDefined()	Метод проверяет, что результат определен
toBeUndefined()	Метод проверяет, что результат не определен
toBeNull()	Метод проверяет, что результат равен null
toBeTruthy()	Метод проверяет, что результат является квазиистинным (см. главу 12)
toBeFalsy()	Метод проверяет, что результат является квазиложным (см. главу 12)
toContain(substring)	Метод проверяет, что результат содержит заданную подстроку
toBeLessThan(value)	Метод проверяет, что результат меньше заданного значения
toBeGreaterThan(value)	Метод проверяет, что результат больше заданного значения

В листинге 29.8 продемонстрировано применение этих методов в тестах, с заменой непроходящего теста из предыдущего раздела.

Листинг 29.8. Замена модульного теста в файле frameworkTest.spec.ts

```
describe("Jasmine Test Environment", () => {
    it("test numeric value", () => expect(12).toBeGreaterThan(10));
    it("test string value", () => expect("London").toMatch("^Lon"));
});
```

При сохранении изменений в файле тесты будут выполнены, а результаты выводятся в командной строке.

Тестирование компонентов Angular

Структурные блоки приложений Angular не могут тестироваться в изоляции, потому что они зависят от базовой функциональности, предоставляемой Angular и другими частями проекта, включая содержащиеся в нем службы, директивы, шаблоны и модули. Как следствие, при тестировании структурного блока, например компонента, приходится пользоваться средствами тестирования, предоставляемыми Angular. Эти средства моделируют приложение в достаточной степени для функционирования компонента, чтобы к нему можно было применять тесты. В этом разделе рассматривается процесс выполнения модульного теста для класса `FirstComponent` функционального модуля `OnDemand`, включенного в проект в главе 27. Напомним, как выглядит определение компонента:

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

@Component({
    selector: "first",
    template: `<div class="bg-primary p-a-1">First Component</div>`
})
export class FirstComponent { }
```

Компонент настолько прост, что он не имеет собственной функциональности для тестирования. Тем не менее и это позволит показать, как проходит процесс тестирования.

Работа с классом TestBed

В области модульного тестирования Angular центральное место занимает класс `TestBed`, отвечающий за моделирование среды приложений Angular для выполнения тестов. В табл. 29.5 описаны самые полезные методы класса `TestBed`. Все эти методы являются статическими (см. главу 6).

Таблица 29.5. Полезные методы TestBed

Имя	Описание
configureTestingModule	Метод используется для настройки тестового модуля Angular
createComponent	Метод используется для создания экземпляра компонента
compileComponents	Метод используется для компиляции компонентов (см. раздел «Тестирование компонента с внешним шаблоном»)

Метод `configureTestingModule` используется для настройки модуля Angular, задействуемого при тестировании, с теми же свойствами, которые поддерживаются декоратором `@NgModel`. Как и в реальных приложениях, компонент не может использоваться в модульном тесте, если он не был добавлен в свойство `declarations` модуля. Это означает, что первым шагом большинства модульных тестов должна стать настройка тестового модуля. Создайте файл first.component.spec.ts в папке tests и включите в него код из листинга 29.9.

Листинг 29.9. Содержимое файла first.component.spec.ts в папке tests

```
import { TestBed } from "@angular/core/testing";
import { FirstComponent } from "../app/ondemand/first.component";

describe("FirstComponent", () => {
    beforeEach(() => {
        TestBed.configureTestingModule({
            declarations: [FirstComponent]
        });
    });
});
```

Класс `TestBed` определяется в модуле `@angular/core/testing`. Метод `configureTestingModule` получает объект, свойство `declarations` которого сообщает тестовому модулю об использовании модуля `FirstComponent`.

ПРИМЕЧАНИЕ

Обратите внимание на использование класса TestBed в функции beforeEach. Если вы попытаетесь использовать TestBed за пределами функции, то получите ошибку об использовании Promise.

На следующем шаге создается новый экземпляр компонента для его использования в тестах. Для этой цели применяется метод `createComponent` (листинг 29.10).

Листинг 29.10. Создание экземпляра компонента в файле first.component.spec.ts

```
import { TestBed, ComponentFixture} from "@angular/core/testing";
import { FirstComponent } from "../app/ondemand/first.component";

describe("FirstComponent", () => {

    let fixture: ComponentFixture<FirstComponent>;
    let component: FirstComponent;

    beforeEach(() => {
        TestBed.configureTestingModule({
            declarations: [FirstComponent]
        });
        fixture = TestBed.createComponent(FirstComponent);
        component = fixture.componentInstance;
    });

    it("is defined", () => {
        expect(component).toBeDefined()
    });
});
```

Аргумент метода `createComponent` сообщает `TestBed`, экземпляр какого типа компонента следует создать; в данном случае это `FirstComponent`. Результатом будет объект `ComponentFixture<FirstComponent>`, который предоставляет функциональность для тестирования компонента с использованием методов и свойств, перечисленных в табл. 29.6.

Таблица 29.6. Полезные методы и свойства ComponentFixture

Имя	Описание
componentInstance	Свойство возвращает объект компонента
debugElement	Свойство возвращает тестовый управляющий элемент для компонента
nativeElement	Свойство возвращает объект DOM, представляющий управляющий элемент для компонента
detectChanges()	Метод заставляет тестовую среду обнаруживать изменения состояния и отражать их в шаблоне компонента
whenStable()	Метод возвращает объект Promise, обрабатываемый при полном применении эффекта операции. За подробностями обращайтесь к разделу «Тестирование с асинхронными операциями»

В листинге свойство `componentInstance` используется для получения объекта `FirstComponent`, созданного тестовой средой, и выполнения простого теста. Метод `expect` выбирает объект компонента как цель теста, а метод `toBeDefined` выполняет тест. Другие методы и свойства будут продемонстрированы в следующих разделах.

ПРИМЕЧАНИЕ

Исходная настройка конфигурации класса TestBed выполняется в файле karma-test-shim.js, показанном в листинге 29.5.

Настройка TestBed для работы с зависимостями

Одной из важнейших особенностей приложений Angular является внедрение зависимостей, позволяющее компонентам и другим структурным блокам получать службы посредством объявления зависимостей от них через параметры конструктора. В листинге 29.11 в класс `FirstComponent` добавляется зависимость для службы репозитория модели данных.

Листинг 29.11. Добавление зависимости для службы в файле first.component.ts

```
import { Component } from "@angular/core";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";

@Component({
    selector: "first",
    template: `<div class="bg-primary p-a-1">
                There are
                    <span class="strong"> {{getProducts().length}} </span>
                products
               </div>`
})
export class FirstComponent {
```

```
    constructor(private repository: Model) {}

    category: string = "Soccer";

    getProducts(): Product[] {
        return this.repository.getProducts()
            .filter(p => p.category == this.category);
    }
}
```

Компонент использует репозиторий для получения отфильтрованной коллекции объектов `Product`, которая читается методом `getProducts` и фильтруется по свойству `category`. Встроенный шаблон содержит соответствующую привязку данных для вывода количества товаров, возвращенных методом `getProducts`.

Для проведения модульного тестирования компонента необходимо предоставить ему службу репозитория. Тестовая среда Angular возьмет на себя разрешение зависимостей при условии, что они были настроены через тестовый модуль. Эффективное модульное тестирование обычно требует изоляции компонентов от остальных частей приложения; это означает, что реальные службы в модульных тестах заменяются *фиктивными* объектами. В листинге 29.12 тестовая среда настраивается таким образом, чтобы фиктивный репозиторий предоставлял компоненту необходимую службу.

Листинг 29.12. Предоставление службы в файле first.component.spec.ts

```
import { TestBed, ComponentFixture} from "@angular/core/testing";
import { FirstComponent } from "../app/ondemand/first.component";
import { Product } from "../app/model/product.model";
import { Model } from "../app/model/repository.model";

describe("FirstComponent", () => {

    let fixture: ComponentFixture<FirstComponent>;
    let component: FirstComponent;

    let mockRepository = {
        getProducts: function () {
            return [
                new Product(1, "test1", "Soccer", 100),
                new Product(2, "test2", "Chess", 100),
                new Product(3, "test3", "Soccer", 100),
            ]
        }
    }

    beforeEach(() => {
        TestBed.configureTestingModule({
            declarations: [FirstComponent],
            providers: [
                { provide: Model, useValue: mockRepository }
            ]
        });
        fixture = TestBed.createComponent(FirstComponent);
```

```
        component = fixture.componentInstance;
    });

    it("filters categories", () => {
        component.category = "Chess"
        expect(component.getProducts().length).toBe(1);
        component.category = "Soccer";
        expect(component.getProducts().length).toBe(2);
        component.category = "Running";
        expect(component.getProducts().length).toBe(0);
    });
});
```

Переменной `mockRepository` присваивается объект, предоставляющий метод `getProducts`, который возвращает фиксированные данные для тестирования известных результатов. Чтобы компонент получил службу, свойство `providers` для объекта, передаваемого методу `TestBed.configureTestingModule`, настраивается по аналогии с реальными модулями Angular: провайдер значения используется для разрешения зависимостей от класса `Model` с использованием переменной `mockRepository`. Тест вызывает метод `getProducts` компонента и сравнивает результаты с ожидаемыми, изменяя значение свойства `category` для проверки разных фильтров.

Тестирование привязок данных

Предыдущий пример показывает, как использовать свойства и методы компонента в модульном тесте. Это хорошее начало, но многие компоненты также включают небольшие фрагменты функциональности в выражениях привязки данных, содержащихся в их шаблонах; их тоже необходимо протестировать. Листинг 29.13 проверяет, что привязка данных в шаблоне компонента правильно выводит количество товаров в фиктивной модели данных.

Листинг 29.13. Модульное тестирование привязки данных в файле first.component.spec.ts

```
import { TestBed, ComponentFixture} from "@angular/core/testing";
import { FirstComponent } from "../app/ondemand/first.component";
import { Product } from "../app/model/product.model";
import { Model } from "../app/model/repository.model";
import { DebugElement } from "@angular/core";
import { By } from "@angular/platform-browser";

describe("FirstComponent", () => {

    let fixture: ComponentFixture<FirstComponent>;
    let component: FirstComponent;
    let debugElement: DebugElement;
    let bindingElement: HTMLSpanElement;

    let mockRepository = {
        getProducts: function () {
            return [
                new Product(1, "test1", "Soccer", 100),
```

```
                new Product(2, "test2", "Chess", 100),
                new Product(3, "test3", "Soccer", 100),
            ]
        }
    }

    beforeEach(() => {
        TestBed.configureTestingModule({
            declarations: [FirstComponent],
            providers: [
                { provide: Model, useValue: mockRepository }
            ]
        });
        fixture = TestBed.createComponent(FirstComponent);
        component = fixture.componentInstance;
        debugElement = fixture.debugElement;
        bindingElement = debugElement.query(By.css("span")).nativeElement;
    });

    it("filters categories", () => {
        component.category = "Chess"
        fixture.detectChanges();
        expect(component.getProducts().length).toBe(1);
        expect(bindingElement.textContent).toContain("1");

        component.category = "Soccer";
        fixture.detectChanges();
        expect(component.getProducts().length).toBe(2);
        expect(bindingElement.textContent).toContain("2");

        component.category = "Running";
        fixture.detectChanges();
        expect(component.getProducts().length).toBe(0);
        expect(bindingElement.textContent).toContain("0");
    });
});
```

Свойство `ComponentFixture.debugElement` возвращает объект `DebugElement`, представляющий корневой элемент из шаблона компонента. В табл. 29.7 перечислены самые полезные методы и свойства класса `DebugElement`.

Таблица 29.7. Полезные методы и свойства DebugElement

Имя	Описание
nativeElement	Свойство возвращает объект, представляющий элемент HTML в DOM
children	Свойство возвращает массив объектов DebugElement, представляющих дочерние элементы этого элемента
query(selectorFunction)	Функции selectorFunction передается объект DebugElement для каждого элемента HTML в шаблоне компонента; метод возвращает первый объект DebugElement, для которого функция возвращает true

Имя	Описание
queryAll(selectorFunction)	Аналог метода query, не считая того, что результат состоит из всех объектов DebugElement, для которых функция возвращает true
triggerEventHandler(name, event)	Метод инициирует событие. За подробностями обращайтесь к разделу «Тестирование событий компонентов»

Поиск элементов осуществляется методами `query` и `queryAll`. Эти методы получают функции, которые проверяют объекты `DebugElement` и возвращают `true` для объектов, включаемых в результаты. Класс `By`, определяемый в модуле `@angular/platform-browser`, упрощает нахождение элементов в шаблоне компонента при помощи статических методов, описанных в табл. 29.8.

Таблица 29.8. Методы By

Имя	Описание
By.all()	Метод возвращает функцию, которая выбирает все элементы
By.css(selector)	Метод возвращает функцию, которая использует селектор CSS для выбора элементов
By.directive(type)	Метод возвращает функцию для выбора элементов, к которым был применен заданный класс директивы (см. раздел «Тестирование входных свойств»)

В листинге метод `By.css` используется для поиска первого элемента `span` в шаблоне и обращения к представляющему его объекту DOM через свойство `nativeElement`. Это позволяет нам проверять значение свойства `textContent` в модульных тестах.

Обратите внимание: после каждого изменения свойства `category` компонента вызывается метод `detectChanges` объекта `ComponentFixture`:

```
...
component.category = "Soccer";
fixture.detectChanges();
expect(component.getProducts().length).toBe(2);
expect(bindingElement.textContent).toContain("2");
...
```

Этот метод приказывает среде тестирования Angular обработать любые изменения и пересчитать выражения привязки данных в шаблоне. Без этого вызова изменение значения `category` в компоненте не будет отражено в шаблоне и тест завершится неудачей.

Тестирование компонента с внешним шаблоном

Компоненты Angular компилируются в классы-фабрики — в браузере или AOT-компилятором (см. главу 10). При этом Angular обрабатывает любые внешние шаблоны и включает в текстовом виде в генерируемый код JavaScript (по аналогии со встроенными шаблонами).

При модульном тестировании компонента с внешним шаблоном шаг компиляции должен выполняться явно. В листинге 29.14 декоратор `@Component`, примененный к классу `FirstComponent`, изменяется с назначением внешнего шаблона.

Листинг 29.14. Назначение внешнего шаблона в файле first.component.ts

```
import { Component } from "@angular/core";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";

@Component({
    selector: "first",
    moduleId: module.id,
    templateUrl: "first.component.html"
})
export class FirstComponent {

    constructor(private repository: Model) {}

    category: string = "Soccer";

    getProducts(): Product[] {
        return this.repository.getProducts()
            .filter(p => p.category == this.category);
    }
}
```

Чтобы определить шаблон, создайте файл first.component.html в папке exampleApp/app/ondemand и добавьте элементы из листинга 29.15.

Листинг 29.15. Файл first.component.html в папке exampleApp/app/ondemand

```
<div class="bg-primary p-a-1">
    There are
        <span class="strong"> {{getProducts().length}} </span>
    products
</div>
```

Это тот же контент, который прежде определялся как встроенный. В листинге 29.16 модульный тест компонента обновляется для перехода на внешний шаблон, что требует явной компиляции компонента.

Листинг 29.16. Компиляция компонента в файле first.component.spec.ts

```
import { TestBed, ComponentFixture, async } from "@angular/core/testing";
import { FirstComponent } from "../app/ondemand/first.component";
import { Product } from "../app/model/product.model";
import { Model } from "../app/model/repository.model";
import { DebugElement } from "@angular/core";
import { By } from "@angular/platform-browser";

describe("FirstComponent", () => {
```

```
    let fixture: ComponentFixture<FirstComponent>;
    let component: FirstComponent;
    let debugElement: DebugElement;
    let spanElement: HTMLSpanElement;

    let mockRepository = {
        getProducts: function () {
            return [
                new Product(1, "test1", "Soccer", 100),
                new Product(2, "test2", "Chess", 100),
                new Product(3, "test3", "Soccer", 100),
            ]
        }
    }

    beforeEach(async(() => {
        TestBed.configureTestingModule({
            declarations: [FirstComponent],
            providers: [
                { provide: Model, useValue: mockRepository }
            ]
        });
        TestBed.compileComponents().then(() => {
            fixture = TestBed.createComponent(FirstComponent);
            component = fixture.componentInstance;
            debugElement = fixture.debugElement;
            spanElement = debugElement.query(By.css("span")).nativeElement;
        });
    }));

    it("filters categories", () => {
        component.category = "Chess"
        fixture.detectChanges();
        expect(component.getProducts().length).toBe(1);
        expect(spanElement.textContent).toContain("1");
    });
});
```

Компоненты компилируются методом `TestBed.compileComponents`. Процесс компиляции работает асинхронно, а метод `compileComponents` возвращает объект `Promise`, который должен использоваться для окончательной настройки теста после завершения компиляции. Чтобы упростить работу с асинхронными операциями в модульных тестах, в модуль `@angular/core/testing` включается функция `async`, которая используется с методом `beforeEach`.

Тестирование событий компонентов

Чтобы показать, как в программах тестируется реакция компонентов на события, мы определим в классе `FirstComponent` новое свойство и добавим метод, к которому применяется декоратор `@HostBinding` (листинг 29.17).

Листинг 29.17. Обработка события в файле first.component.ts

```
import { Component, HostListener} from "@angular/core";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";

@Component({
    selector: "first",
    moduleId: module.id,
    templateUrl: "first.component.html"
})
export class FirstComponent {

    constructor(private repository: Model) {}

    category: string = "Soccer";
    highlighted: boolean = false;

    getProducts(): Product[] {
        return this.repository.getProducts()
            .filter(p => p.category == this.category);
    }

    @HostListener("mouseenter", ["$event.type"])
    @HostListener("mouseleave", ["$event.type"])
    setHighlight(type: string) {
        this.highlighted = type == "mouseenter";
    }
}
```

Метод `setHighlight` настраивается таким образом, чтобы он вызывался при инициировании событий `mouseenter` и `mouseleave` управляющего элемента. В листинге 29.18 шаблон компонента обновляется так, чтобы новое свойство использовалось в привязке данных.

Листинг 29.18. Привязка для свойства в файле first.component.html

```
<div class="bg-primary p-a-1" [class.bg-success]="highlighted">
    There are
        <span class="strong"> {{getProducts().length}} </span>
    products
</div>
```

Для инициирования событий в модульных тестах можно воспользоваться методом `triggerEventHandler`, определенным классом `DebugElement` (листинг 29.19).

Листинг 29.19. Инициирование событий в файле first.component.spec.ts

```
import { TestBed, ComponentFixture, async } from "@angular/core/testing";
import { FirstComponent } from "../app/ondemand/first.component";
import { Product } from "../app/model/product.model";
import { Model } from "../app/model/repository.model";
import { DebugElement } from "@angular/core";
import { By } from "@angular/platform-browser";

describe("FirstComponent", () => {
```

```
    let fixture: ComponentFixture<FirstComponent>;
    let component: FirstComponent;
    let debugElement: DebugElement;
    let divElement: HTMLDivElement;

    let mockRepository = {
        getProducts: function () {
            return [
                new Product(1, "test1", "Soccer", 100),
                new Product(2, "test2", "Chess", 100),
                new Product(3, "test3", "Soccer", 100),
            ]
        }
    }

    beforeEach(async(() => {
        TestBed.configureTestingModule({
            declarations: [FirstComponent],
            providers: [
                { provide: Model, useValue: mockRepository }
            ]
        });
        TestBed.compileComponents().then(() => {
            fixture = TestBed.createComponent(FirstComponent);
            component = fixture.componentInstance;
            debugElement = fixture.debugElement;
            divElement = debugElement.children[0].nativeElement;
        });
    }));

    it("handles mouse events", () => {
        expect(component.highlighted).toBeFalsy();
        expect(divElement.classList.contains("bg-success")).toBeFalsy();
        debugElement.triggerEventHandler("mouseenter", new Event("mouseenter"));
        fixture.detectChanges();
        expect(component.highlighted).toBeTruthy();
        expect(divElement.classList.contains("bg-success")).toBeTruthy();
        debugElement.triggerEventHandler("mouseleave", new Event("mouseleave"));
        fixture.detectChanges();
        expect(component.highlighted).toBeFalsy();
        expect(divElement.classList.contains("bg-success")).toBeFalsy();
    });
});
```

Тест в этом листинге проверяет исходное состояние компонента и шаблона, а затем инициирует события `mouseenter` и `mouseleave`, проверяя эффект каждого из них.

Тестирование выходных свойств

Тестирование выходных свойств — достаточно простой процесс, потому что объекты `EventEmitter`, используемые для их реализации, являются объектами `Observable`, на которые можно подписываться в модульных тестах. В листинге 29.20 в тестируемый компонент добавляется выходное свойство.

Листинг 29.20. Добавление выходного свойства в файле first.component.ts

```
import { Component, HostListener, Output, EventEmitter} from "@angular/core";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";

@Component({
    selector: "first",
    moduleId: module.id,
    templateUrl: "first.component.html"
})
export class FirstComponent {

    constructor(private repository: Model) {}

    category: string = "Soccer";
    highlighted: boolean = false;

    @Output("pa-highlight")
    change = new EventEmitter<boolean>();

    getProducts(): Product[] {
        return this.repository.getProducts()
            .filter(p => p.category == this.category);
    }

    @HostListener("mouseenter", ["$event.type"])
    @HostListener("mouseleave", ["$event.type"])
    setHighlight(type: string) {
        this.highlighted = type == "mouseenter";
        this.change.emit(this.highlighted);
    }
}
```

Компонент определяет выходное свойства с именем `change`, которое используется для генерирования события при вызове метода `setHighlight`. В листинге 29.21 представлен модульный тест для выходного свойства.

Листинг 29.21. Тестирование выходного свойства в файле first.component.spec.ts

```
import { TestBed, ComponentFixture, async } from "@angular/core/testing";
import { FirstComponent } from "../app/ondemand/first.component";
import { Product } from "../app/model/product.model";
import { Model } from "../app/model/repository.model";
import { DebugElement } from "@angular/core";
import { By } from "@angular/platform-browser";

describe("FirstComponent", () => {

    let fixture: ComponentFixture<FirstComponent>;
    let component: FirstComponent;
    let debugElement: DebugElement;

    let mockRepository = {
        getProducts: function () {
            return [
```

```
                new Product(1, "test1", "Soccer", 100),
                new Product(2, "test2", "Chess", 100),
                new Product(3, "test3", "Soccer", 100),
            ]
        }
    }

    beforeEach(async(() => {
        TestBed.configureTestingModule({
            declarations: [FirstComponent],
            providers: [
                { provide: Model, useValue: mockRepository }
            ]
        });
        TestBed.compileComponents().then(() => {
            fixture = TestBed.createComponent(FirstComponent);
            component = fixture.componentInstance;
            debugElement = fixture.debugElement;
        });
    }));

    it("implements output property", () => {
        let highlighted: boolean;
        component.change.subscribe(value => highlighted = value);
        debugElement.triggerEventHandler("mouseenter", new Event("mouseenter"));
        expect(highlighted).toBeTruthy();
        debugElement.triggerEventHandler("mouseleave", new Event("mouseleave"));
        expect(highlighted).toBeFalsy();
    });
});
```

Метод `setHighlight` компонента также можно было вызвать прямо из модульного теста, но вместо этого я решил инициировать события `mouseenter` и `mouseleave`, которые активизируют выходное свойство косвенно.

Прежде чем инициировать события, я использую метод `subscribe` для получения события от выходного свойства, которое затем используется для проверки ожидаемых событий.

Тестирование входных свойств

Процесс тестирования входных свойств требует небольшой дополнительной работы. Для начала добавьте в класс `FirstComponent` входное свойство, которое используется для получения репозитория модели данных вместо службы, которая передавалась в конструкторе (листинг 29.22). Также удалите привязки событий и входное свойство, чтобы упростить пример.

Листинг 29.22. Добавление входного свойства в файле first.component.ts

```
import { Component, HostListener, Input } from "@angular/core";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
```

```
@Component({
    selector: "first",
    moduleId: module.id,
    templateUrl: "first.component.html"
})
export class FirstComponent {

    category: string = "Soccer";
    highlighted: boolean = false;

    getProducts(): Product[] {
        return this.model == null ? [] : this.model.getProducts()
            .filter(p => p.category == this.category);
    }

    @Input("pa-model")
    model: Model;
}
```

Входное свойство задается при помощи атрибута `pa-model` и используется в методе `getProducts`. В листинге 29.23 показано, как написать модульный тест для входного свойства.

Листинг 29.23. Тестирование входного свойства в файле first.component.spec.ts

```
import { TestBed, ComponentFixture, async } from "@angular/core/testing";
import { FirstComponent } from "../app/ondemand/first.component";
import { Product } from "../app/model/product.model";
import { Model } from "../app/model/repository.model";
import { DebugElement } from "@angular/core";
import { By } from "@angular/platform-browser";
import { Component, ViewChild } from "@angular/core";

@Component({
    template: `<first [pa-model]="model"></first>`
})
class TestComponent {

    constructor(public model: Model) { }
    @ViewChild(FirstComponent)
    firstComponent: FirstComponent;
}

describe("FirstComponent", () => {

    let fixture: ComponentFixture<TestComponent>;
    let component: FirstComponent;
    let debugElement: DebugElement;

    let mockRepository = {
        getProducts: function () {
            return [
```

```
                new Product(1, "test1", "Soccer", 100),
                new Product(2, "test2", "Chess", 100),
                new Product(3, "test3", "Soccer", 100),
            ]
        }
    }

    beforeEach(async(() => {
        TestBed.configureTestingModule({
            declarations: [FirstComponent, TestComponent],
            providers: [
                { provide: Model, useValue: mockRepository }
            ]
        });
        TestBed.compileComponents().then(() => {
            fixture = TestBed.createComponent(TestComponent);
            component = fixture.componentInstance.firstComponent;
            debugElement = fixture.debugElement.query(By.directive(FirstComponent));
        });
    }));

    it("receives the model through an input property", () => {
        component.category = "Chess";
        fixture.detectChanges();
        let products = mockRepository.getProducts()
            .filter(p => p.category == component.category);
        let componentProducts = component.getProducts();
        for (let i = 0; i < componentProducts.length; i++) {
            expect(componentProducts[i]).toEqual(products[i]);
        }
        expect(debugElement.query(By.css("span")).nativeElement.textContent)
            .toContain(products.length);
    });
});
```

Здесь определяется компонент, который нужен только для создания теста; шаблон этого компонента содержит элемент, соответствующий селектору целевого компонента. В данном примере определяется класс компонента с именем `TestComponent` во встроенном шаблоне, определенном в декораторе `@Component`, который содержит первый элемент с атрибутом `pa-model`, соответствующий декоратору `@Input`, примененному к классу `FirstComponent` (листинг 29.22).

Класс тестового компонента добавляется в массив `declarations` модуля тестирования, а его экземпляр создается методом `TestBed.createComponent`. В классе `TestComponent` используется декоратор `@ViewChild`, чтобы мы могли получить экземпляр `FirstComponent`, необходимый для тестирования. Для получения корневого элемента `FirstComponent` используется метод `DebugElement.query` в сочетании с методом `By.directive`.

В результате мы можем обращаться как к компоненту, так и к его корневому элементу из теста, который задает свойство `category`, а затем проверяет результаты — как от компонента, так и через привязку данных в его шаблоне.

Тестирование с асинхронными операциями

Еще одна область, требующая особых мер, — асинхронные операции. Для демонстрации тестирования с асинхронными операциями листинг 29.24 изменяет тестируемый компонент, чтобы для получения данных в нем использовался класс `RestDataSource`, определенный в главе 24. Этот класс не предназначен для использования за пределами функционального модуля модели, но он предоставляет полезный набор асинхронных методов, которые возвращают объекты `Observable`, поэтому я нарушил спланированную структуру приложения для демонстрации методики тестирования.

Листинг 29.24. Выполнение асинхронной операции в файле first.component.ts

```
import { Component, HostListener, Input } from "@angular/core";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
import { RestDataSource } from "../model/rest.datasource";
import { Observable } from "rxjs/Observable";

@Component({
    selector: "first",
    moduleId: module.id,
    templateUrl: "first.component.html"
})
export class FirstComponent {
    _category: string = "Soccer";
    _products: Product[] = [];
    highlighted: boolean = false;

    constructor(public datasource: RestDataSource) {}

    ngOnInit() {
        this.updateData();
    }

    getProducts(): Product[] {
        return this._products;
    }

    set category(newValue: string) {
        this._category;
        this.updateData();
    }

    updateData() {
        this.datasource.getData()
            .subscribe(data => this._products = data
                .filter(p => p.category == this._category));
    }
}
```

Компонент получает свои данные через метод `getData` источника данных, который возвращает объект `Observable`. Компонент подписывается на `Observable` и обнов-

ляет свое свойство `_product` с объектами данных, доступ к которым предоставляется шаблону через метод `getProducts`.

В листинге 29.25 показано, как организуется тестирование таких компонентов с использованием средств Angular для работы с асинхронными операциями в модульных тестах.

Листинг 29.25. Тестирование асинхронных операций в файле first.component.spec.ts

```
import { TestBed, ComponentFixture, async } from "@angular/core/testing";
import { FirstComponent } from "../app/ondemand/first.component";
import { Product } from "../app/model/product.model";
import { Model } from "../app/model/repository.model";
import { DebugElement } from "@angular/core";
import { By } from "@angular/platform-browser";
import { Component, ViewChild } from "@angular/core";
import { RestDataSource } from "../app/model/rest.datasource";
import { Observable } from "rxjs/Observable";
import "rxjs/add/observable/from";
import { Injectable } from "@angular/core";

@Injectable()
class MockDataSource {
    public data = [
        new Product(1, "test1", "Soccer", 100),
        new Product(2, "test2", "Chess", 100),
        new Product(3, "test3", "Soccer", 100),
    ];

    getData(): Observable<Product[]> {
        return new Observable<Product>(obs => {
            setTimeout(() => obs.next(this.data), 1000);
        })
    }
}

describe("FirstComponent", () => {

    let fixture: ComponentFixture<FirstComponent>;
    let component: FirstComponent;
    let dataSource = new MockDataSource();

    beforeEach(async(() => {
        TestBed.configureTestingModule({
            declarations: [FirstComponent],
            providers: [
                { provide: RestDataSource, useValue: dataSource }
            ]
        });
        TestBed.compileComponents().then(() => {
            fixture = TestBed.createComponent(FirstComponent);
            component = fixture.componentInstance;
        });
    }));
```

```
    it("performs async op", () => {
        dataSource.data.push(new Product(100, "test100", "Soccer", 100));
        fixture.detectChanges();

        fixture.whenStable().then(() => {
            expect(component.getProducts().length).toBe(3);
        });
    });
});
```

Фиктивный объект в этом примере имеет более полноценную структуру, чем тот, который создавался ранее, — просто для демонстрации других способов достижения той же цели. Важно заметить, что реализуемый им метод `getData` создает секундную задержку перед возвращением данных.

Задержка важна, так как она означает, что эффект вызова метода `detectChanges` в модульном тесте не оказывает непосредственного влияния на компонент. Для решения этой проблемы я вызываю метод `whenStable`, определенный классом `ComponentFixture`, который возвращает объект `Primise`, разрешаемый при полной обработке всех изменений. Это позволяет отложить проверку результатов теста до того момента, когда объект `Observable`, возвращаемый фиктивным источником данных, доставит данные компоненту.

Тестирование директив Angular

Процесс тестирования директив похож на процесс тестирования входных свойств в том отношении, что тестовый компонент и шаблон используются для создания тестового окружения, к которому может применяться директива. Чтобы у вас была директива для тестирования, создайте файл attr.directive.ts в папке exampleApp/app/ondemand и включите в него код из листинга 29.26.

ПРИМЕЧАНИЕ

В данном примере используется директива атрибута, но механизм, описанный в этом разделе, может применяться и для структурных директив.

Листинг 29.26. Содержимое файла attr.directive.ts в папке exampleApp/app/ondemand

```
import {
    Directive, ElementRef, Attribute, Input, SimpleChange
} from "@angular/core";

@Directive({
    selector: "[pa-attr]"
})

export class PaAttrDirective {

    constructor(private element: ElementRef) { }
```

```
    @Input("pa-attr")
    bgClass: string;

    ngOnChanges(changes: { [property: string]: SimpleChange }) {
        let change = changes["bgClass"];
        let classList = this.element.nativeElement.classList;
        if (!change.isFirstChange() && classList.contains(change.previousValue)) {
            classList.remove(change.previousValue);
        }
        if (!classList.contains(change.currentValue)) {
            classList.add(change.currentValue);
        }
    }
}
```

Эта директива атрибута основана на примере из главы 15. Чтобы создать модульный тест для этой директивы, создайте файл attr.directive.spec.ts в папке exampleApp/tests и включите в него код из листинга 29.27.

Листинг 29.27. Содержимое файла attr.directive.spec.ts в папке exampleApp/tests

```
import { TestBed, ComponentFixture } from "@angular/core/testing";
import { Component, DebugElement, ViewChild } from "@angular/core";
import { By } from "@angular/platform-browser";
import { PaAttrDirective } from "../app/ondemand/attr.directive";

@Component({
    template: `<div><span [pa-attr]="className">Test Content</span></div>`
})
class TestComponent {
    className = "initialClass"

    @ViewChild(PaAttrDirective)
    attrDirective: PaAttrDirective;
}

describe("PaAttrDirective", () => {

    let fixture: ComponentFixture<TestComponent>;
    let directive: PaAttrDirective;
    let spanElement: HTMLSpanElement;

    beforeEach(() => {
        TestBed.configureTestingModule({
            declarations: [TestComponent, PaAttrDirective],
        });

        fixture = TestBed.createComponent(TestComponent);
        directive = fixture.componentInstance.attrDirective;
        spanElement = fixture.debugElement.query(By.css("span")).nativeElement;
    });
    it("generates the correct number of elements", () => {
        fixture.detectChanges();
```

```
        expect(directive.bgClass).toBe("initialClass");
        expect(spanElement.className).toBe("initialClass");

        fixture.componentInstance.className = "nextClass";
        fixture.detectChanges();
        expect(directive.bgClass).toBe("nextClass");
        expect(spanElement.className).toBe("nextClass");
    });
});
```

Тестовый компонент содержит встроенный шаблон, который применяет директиву и свойство, упоминаемое в привязке данных. Декоратор `@ViewChild` предоставляет доступ к объекту директивы, который Angular создает при обработке директивы. Модульный тест может проверить, что изменение значения, используемого привязкой данных, оказывает нужное воздействие на объект директивы и элемент, к которому он был применен.

Итоги

В этой главе продемонстрированы различные способы использования директив и компонентов Angular для модульного тестирования. Вы узнали, как установить тестовую инфраструктуру и инструменты и как создать тестовую среду для применения тестов. Я объяснил, как тестировать различные аспекты компонентов и как применять те же средства для тестирования директив.

И это все, что я могу рассказать вам об Angular. Мы начали с создания простого приложения, а затем занялись долгим и подробным изучением структурных блоков фреймворка; вы узнали, как создавать, настраивать и использовать эти блоки при разработке веб-приложений.

Желаю вам успеха в разработке приложений Angular! Надеюсь, вы получили от чтения книги столько же удовольствия, сколько я — от работы над ней.