Мэтт Фрисби

Angular

Сборник рецептов

Более 70 рецептов с решениями, которые следует знать, столкнувшись с трудностями в Angular 2!

Packt>

Мэтт Фрисби

Angular

Сборник рецептов

Matt Frisbie

Angular 2

Cookbook

Packt>

Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

Мэтт Фрисби

Angular

Сборник рецептов

ДИАЛЕКТИКА

Москва ◆ Санкт-Петербург
2018

ББК 32.973.26-018.2.75
Ф89
УДК 681.3.07

Компьютерное издательство "Диалектика"

По общим вопросам обращайтесь в издательство "Диалектика" по адресу:
info@dialektika.com, http://www.dialektika.com

Фрисби, Мэтт.

Ф89 Angular. Сборник рецептов, 2-е изд.: Пер. с англ. — СПб.: ООО "Альфа-книга", 2018. — 464 с. : ил. — Парал. тит. англ.

ISBN 978-5-9909446-6-4 (рус.)

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства Packt Publishing Ltd.

Authorized Russian translation of the English edition of *Angular 2 Cookbook* (ISBN 978-1-78588-192-3) © 2017 Packt Publishing. All rights reserved.

This translation is published and sold by permission of Packt Publishing Ltd., which owns or controls all rights to publish and sell the same.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the Publisher.

Научно-популярное издание

Мэтт Фрисби

Angular. Сборник рецептов

2-е издание

ООО "Альфа-книга", 195027, Санкт-Петербург, Магнитогорская ул., д. 30, лит. А, пом. 848

ISBN 978-5-9909446-6-4 (рус.)
ISBN 978-1-7858819213 (англ.)

© Компьютерное издательство "Диалектика", 2018
© Packt Publishing, 2017

Оглавление

Предисловие 22

Глава 1

СТРАТЕГИИ ПЕРЕХОДА К ANGULAR 2 27

Глава 2

ЗНАКОМСТВО С КОМПОНЕНТАМИ И ДИРЕКТИВАМИ 55

Глава 3

СОЗДАНИЕ РЕАКТИВНЫХ И УПРАВЛЯЕМЫХ ПО ШАБЛОНАМ ФОРМ 115

Глава 4

ИСПОЛЬЗОВАНИЕ ОБЯЗАТЕЛЬСТВ 159

Глава 5

НАБЛЮДАЕМЫЕ ОБЪЕКТЫ ИЗ БИБЛИОТЕКИ REACTIVEX 187

Глава 6

МОДУЛЬ COMPONENT ROUTER 231

Глава 7

СЛУЖБЫ, ВНЕДРЕНИЕ ЗАВИСИМОСТЕЙ И КЛАСС `NgModule` 279

Глава 8

ОРГАНИЗАЦИЯ ПРИЛОЖЕНИЙ И УПРАВЛЕНИЕ ИМИ 311

Глава 9

ТЕСТИРОВАНИЕ В ANGULAR 2 349

Глава 10

ПРОИЗВОДИТЕЛЬНОСТЬ И РАЗВИТЫЕ КОНЦЕПЦИИ 395

Предметный указатель 447

Содержание

Об авторе 20
О рецензенте 20
Благодарности 21

Предисловие 22
О чем эта книга? 22
Что требуется для этой книги 24
Кому адресована эта книга 24
Условные обозначения, принятые в книге 25
Загрузка исходного кода примеров 26
Ждем ваших отзывов! 26

Глава 1. Стратегии перехода к Angular 2 27

Введение 27
Разбиение директив на компоненты путем инкапсуляции в свойстве `controllerAs` 28
Подготовка 28
Реализация 29
Принцип действия 32
Дополнение 32
См. также 33
Перенос приложения в директивы компонентов 33
Подготовка 33
Реализация 34
Принцип действия 36
Дополнение 37
См. также 37
Реализация основного компонента в версии AngularJS 1.5 37
Подготовка 38
Реализация 38
Принцип действия 39
Дополнение 40
См. также. 41
Нормализация типов служб 41
Подготовка 41
Реализация 42
Принцип действия 43
Дополнение 43
См. также 43

Связывание Angular 1 и Angular 2 средствами класса `UpgradeModule` 44
Подготовка 44
Реализация 44
Связывание Angular 1 и Angular 2 46
Принцип действия 47
Дополнение 47
См. также 48
Обратный переход от компонентов Angular 2 к директивам Angular 1 с помощью функции `downgradeComponent()` 48
Подготовка 48
Реализация 49
Принцип действия 51
См. также 52
Обратный переход от поставщиков Angular 2 к службам Angular 1 с помощью функции `downgradeInjectable()` 52
Подготовка 52
Реализация 53
См. также 54

Глава 2. Знакомство с компонентами и директивами 55

Введение 56
Применение декораторов для создания и стилизации простого компонента 56
Подготовка 56
Реализация 57
Принцип действия 60
Дополнение 61
См. также 62
Передача членов родительского компонента порожденному компоненту 62
Подготовка 62
Реализация 63
Принцип действия 67
Дополнение 67
См. также 69
Привязка к естественным атрибутам элементов разметки 70
Реализация 70
Принцип действия 70
См. также 71
Регистрация обработчиков естественных событий в браузерах 71
Подготовка 71
Реализация 72
Принцип действия 72

Дополнение 73
См. также 74
Генерирование и обработка специальных событий средствами класса `EventEmitter` 74
Подготовка 74
Реализация 75
Принцип действия 78
Дополнение 79
См. также 79
Присоединение поведения к элементам модели DOM с помощью директив 79
Подготовка 80
Реализация 80
Принцип действия 83
Дополнение 83
См. также 83
Отображение вложенного содержимого с помощью директивы `ngContent` 84
Подготовка 84
Реализация 85
Принцип действия 86
Дополнение 86
См. также 87
Применение структурных директив `ngFor` и `ngIf` для управления на основе модели DOM 87
Подготовка 87
Реализация 88
Принцип действия 89
Дополнение 90
См. также 91
Обращение к элементам разметки с помощью переменных шаблона 91
Подготовка 92
Реализация 92
Дополнение 93
См. также 94
Привязка свойств атрибутов 95
Подготовка 95
Реализация 96
Принцип действия 97
Дополнение 98
См. также 98
Применение перехватчиков жизненного цикла компонентов 98
Подготовка 98
Реализация 99

Принцип действия 100
Дополнение 101
См. также 101
Обращение к родительскому компоненту из порожденного компонента 101
Подготовка 101
Реализация 102
Принцип действия 103
Дополнение 103
См. также 103
Настройка взаимного оповещения родительских и порожденных компонентов с помощью декоратора `@ViewChild` и функции `forwardRef()` 104
Подготовка 104
Реализация 105
Принцип действия 107
Дополнение 108
См. также 109
Настройка взаимного оповещения родительских и порожденных компонентов с помощью декоратора `@ContentChild` и функции `forwardRef()` 109
Подготовка 110
Реализация 110
Принцип действия 112
Дополнение 113
См. также 114

Глава 3. Создание реактивных и управляемых по шаблонам форм 115

Введение 115
Реализация простой двунаправленной привязки данных с помощью директивы `ngModel` 116
Реализация 116
Принцип действия 117
Дополнение 118
См. также 119
Реализация элементарной проверки достоверности данных в поле средствами класса `FormControl` 119
Подготовка 120
Реализация 120
Принцип действия 122
Дополнение 123
См. также 124
Объединение объектов типа `FormControl` средствами класса `FormGroup` 124

Подготовка 125
Реализация 125
Принцип действия 128
Дополнение 128
См. также 129
Объединение объектов типа FormControl средствами класса FormArray 130
Подготовка 130
Реализация 130
Принцип действия 132
Дополнение 133
См. также 134
Реализация элементарных форм с помощью директивы ngForm 134
Подготовка 134
Реализация 135
Принцип действия 138
Дополнение 139
См. также 140
Реализация элементарных форм средствами класса FormBuilder и директивы formControlName 140
Подготовка 140
Реализация 141
Принцип действия 144
Дополнение 144
См. также 145
Создание и применение специального средства проверки достоверности 146
Подготовка 146
Реализация 147
Принцип действия 148
Дополнение 149
См. также 153
Создание и применение асинхронного средства проверки достоверности с помощью обязательств 153
Подготовка 154
Реализация 154
Принцип действия 156
Дополнение 157
См. также 157

Глава 4. Использование обязательств 159

Введение 159
Представление и реализация основных обязательств 160
Подготовка 160
Реализация 161

Принцип действия 164
Дополнение 165
См. также 168
Связывание обязательств и их обработчиков в цепочку 168
Реализация 169
Принцип действия 170
Дополнение 170
См. также 172
Создание оболочек для обязательств с помощью методов `Promise.resolve()` и `Promise.reject()` 172
Реализация 173
Принцип действия 174
Дополнение 174
См. также 175
Реализация барьеров для обязательств с помощью метода `Promise.all()` 175
Реализация 175
Принцип действия 176
Дополнение 177
См. также 177
Отмена асинхронных действий с помощью метода `Promise.race()` 178
Подготовка 179
Реализация 179
Принцип действия 180
См. также 181
Преобразование обязательств в наблюдаемые объекты 181
Реализация 181
Принцип действия 182
Дополнение 183
См. также 183
Преобразование наблюдаемого объекта HTTP-службы в обязательство типа `ZoneAwarePromise` 183
Подготовка 184
Реализация 184
Принцип действия 185
См. также 185

Глава 5. Наблюдаемые объекты из библиотеки ReactiveX 187

Введение 187
Шаблон "Наблюдатель" 188
Библиотеки ReactiveX и RxJS 188
Наблюдаемые объекты в Angular 2 188
Наблюдаемые объекты и обязательства 189

Простое применение наблюдаемых объектов в HTTP-запросах 189
Подготовка 189
Реализация 190
Принцип действия 192
Дополнение 194
См. также 194
Реализация модели "издатель–подписчик" средствами класса `Subject` 194
Подготовка 195
Реализация 195
Принцип действия 197
Дополнение 197
См. также 199
Создание наблюдаемой службы аутентификации средствами класса `BehaviorSubject` 200
Подготовка 200
Реализация 201
Принцип действия 205
Дополнение 206
См. также 207
Создание обобщенной службы "издатель–подписчик" для замены служб `$broadcast`, `$emit` и `$on` 207
Подготовка 208
Реализация 209
Принцип действия 213
Дополнение 214
См. также 215
Отслеживание изменений в декораторе `@ViewChildren` средствами классов `QueryList` и `Observable` 216
Подготовка 216
Реализация 217
Принцип действия 221
См. также 221
Создание полноценного компонента `AutoComplete` с помощью наблюдаемых объектов 221
Подготовка 222
Реализация 223
Принцип действия 229
См. также 229

Глава 6. Модуль Component Router 231

Введение 231
Настройка приложения на поддержку простых маршрутов 232
Подготовка 232

Реализация 232
Принцип действия 236
Дополнение 237
См. также 238
Навигация по ссылкам с помощью директивы routerLink 238
Подготовка 238
Принцип действия 239
Принцип действия 240
Дополнение 241
См. также 243
Навигация с помощью службы Router 243
Подготовка 243
Принцип действия 243
Принцип действия 245
Дополнение 245
См. также 245
Выбор стратегии обнаружения ресурсов типа LocationStrategy для построения пути 245
Реализация 246
Дополнение 247
Реализация сохраняющего состояние маршрута с помощью директивы routerLinkActive 248
Подготовка 248
Реализация 248
Принцип действия 250
Дополнение 250
См. также 251
Реализация вложенных представлений с помощью параметров маршрута и порожденных маршрутов 252
Подготовка 252
Реализация 252
Принцип действия 257
Дополнение 257
См. также 258
Обработка матричных параметров URL и массивов маршрутизации 259
Подготовка 259
Реализация 259
Принцип действия 262
Дополнение 262
См. также 262
Добавление элементов управления аутентификацией маршрутов с помощью ограничителей маршрутов 263
Подготовка 263
Реализация 263

Принцип действия 276
Дополнение 277
См. также 277

Глава 7. Службы, внедрение зависимостей и класс `NgModule` 279

Введение 279
Внедрение простой службы в компонент 280
Подготовка 280
Реализация 280
Принцип действия 283
Дополнение 283
См. также 284
Контроль над созданием экземпляра службы и внедрением средствами класса `NgModule` 284
Подготовка 284
Реализация 285
Принцип действия 288
Дополнение 288
См. также 290
Назначение псевдонимов при внедрении служб с помощью свойств `useClass` и `useExisting` 290
Подготовка 291
Реализация 293
Принцип действия 295
Дополнение 295
См. также 299
Внедрение значения в виде службы с помощью свойства `useValue` и класса `OpaqueToken` 300
Подготовка 300
Реализация 301
Принцип действия 303
Дополнение 303
См. также 305
Создание настраиваемой поставщиком службы с помощью свойства `useFactory` 305
Подготовка 306
Реализация 306
Принцип действия 309
Дополнение 309
См. также 309

Глава 8. Организация приложений и управление ими 311

Введение 311
Содержимое файла `package.json` для минимально жизнеспособного приложения Angular 2 312
Подготовка 312
Реализация 313
См. также 317
Конфигурирование TypeScript для минимально жизнеспособного приложения Angular 2 317
Подготовка 318
Реализация 318
Принцип действия 320
Дополнение 322
См. также 324
Транспиляция в браузере средствами библиотеки SystemJS 324
Подготовка 324
Реализация 325
Принцип действия 326
Дополнение 326
См. также 326
Составление файлов приложений для минимально жизнеспособного приложения Angular 2 327
Подготовка 327
Реализация 327
См. также 332
Перенос минимально жизнеспособного приложения Angular 2 в сборку Webpack 333
Подготовка 333
Реализация 334
См. также 337
Внедрение прокладок и полифилов в Webpack 337
Подготовка 337
Реализация 337
Принцип действия 338
См. также 338
Формирование HTML-разметки с помощью подключаемого модуля `html-webpack-plugin` 339
Подготовка 339
Реализация 339
Принцип действия 341
См. также 341
Установка приложения Angular с помощью интерфейса командной строки 341

Подготовка 342
Реализация 342
Принцип действия 343
Дополнение 346
См. также 347

Глава 9. Тестирование в Angular 2 349

Введение 349
Создание минимально жизнеспособного набора модульных тестов средствами Karma, Jasmine и TypeScript 350
Подготовка 351
Реализация 351
Принцип действия 357
Дополнение 357
См. также 357
Написание минимально жизнеспособного набора модульных тестов для простого компонента 358
Подготовка 358
Реализация 359
Принцип действия 365
См. также 365
Написание минимально жизнеспособного набора сквозных тестов для простого приложения 366
Подготовка 366
Реализация 367
Принцип действия 373
Дополнение 374
См. также 374
Модульное тестирование синхронной службы 374
Подготовка 375
Реализация 376
Принцип действия 379
Дополнение 379
См. также 380
Модульное тестирование зависимого от службы компонента с помощью заглушек 381
Подготовка 381
Реализация 382
Реализация 386
См. также 386
Модульное тестирование зависимого от службы компонента с помощью шпионов 386
Подготовка 387

Принцип действия 388
Принцип действия 392
Дополнение 392
См. также 393

Глава 10. Производительность и развитые концепции 395

Введение 395
Общее представление о функции `enableProdMode()` и надлежащем ее применении в чистых и нечистых каналах 396
Подготовка 396
Реализация 397
Принцип действия 400
Дополнение 400
См. также 401
Работа с зонами за пределами Angular 401
Подготовка 402
Реализация 403
Принцип действия 408
Дополнение 408
См. также 409
Прием событий от службы типа `NgZone` 410
Подготовка 410
Реализация 411
Принцип действия 412
См. также 414
Выполнение за пределами зоны Angular 414
Реализация 414
Принцип действия 417
Дополнение 417
См. также 417
Настройка компонентов на явное обнаружение изменений с помощью стратегии `OnPush` 418
Подготовка 418
Реализация 419
Принцип действия 421
Дополнение 421
См. также 422
Настройка инкапсуляции представлений на максимальную производительность 422
Подготовка 423
Реализация 423
Принцип действия 426

Дополнение 426
См. также 426
Настройка службы типа `Renderer` на применение рабочих веб-процессов 426
Подготовка 427
Реализация 428
Принцип действия 430
Дополнение 431
См. также 432
Настройка приложений на компиляцию перед выполнением 432
Подготовка 432
Реализация 433
Принцип действия 435
Дополнение 436
См. также 437
Настройка приложений на загрузку по требованию 437
Подготовка 437
Реализация 439
Принцип действия 441
Дополнение 441
См. также 445

Предметный указатель 447

Моему деду Ричарду и бабушке Марджери, отстаивавших честь семьи.

Об авторе

Мэтт Фрисби занимается в настоящее время разработкой программного обеспечения в компании Google. Он является автором книги *AngularJS Web Application Development Cookbook*, вышедшей в издательстве Packt Publishing, а также серии учебных видеоматериалов, опубликованных в издательстве O'Reilly. Он принимает активное участие в сообществе разработчиков и пользователей Angular, делая презентации на семинарах и организуя прямые веб-трансляции в оперативном режиме.

О рецензенте

Патрик Гиллеспи (Patrick Gillespie) занимается разработкой программного обеспечения с 1996 года, имеет степень бакалавра и магистра в области вычислительной техники. В свободное от работы время он увлекается фотографией, проводит досуг в своей семье и работает над различными сторонними проектами для своего веб-сайта (`http://patorjk.com/`).

Благодарности

Написать книгу по Angular 2, когда этот каркас еще не был завершен, оказалось непростым делом. Разрозненные примеры, неоконченная документация и только зарождающееся сообщество разработчиков Angular — это лишь часть многочисленных препятствий, стоявших на пути к написанию данной книги. Ее удалось завершить только благодаря поддержке многочисленной армии приверженцев Angular 2.

Эта книга увидела свет благодаря неустанному труду всего коллектива издательства Packt Publishing Ltd. И в этой связи особой благодарности заслуживают Арун Надар (Arun Nadar), Вивек Арора (Vivek Arora), Мервин д'Соуза (Merwyn D'Souza) и Виней Аргекар (Vinay Argekar) — за их редакторскую работу и опеку, а также Патрик Гиллеспи — за рецензирование книги. Хотелось бы также поблагодарить Джордана, Зою, Скотта, моих родных и друзей за оказанную моральную поддержку в работе над книгой.

Предисловие

"Все что-то планируют до тех пор, пока не получат по зубам".

Майк Тайсон, абсолютный чемпион по боксу є тяжелом весе

Как только каркас AngularJS был создан в 2009 году, он получил широкое распространение как инструментальное средство для написания клиентских приложений. С годами, когда стали появляться новые зыпуски и постепенно созревало сообщество программирующих на JavaScript, область разработки клиентских приложений расширилась за те пределы, для которых был предназначен каркас AngularJS. Опекавшие его критически оценили создавшееся положение и решили, что настало время переделать весь каркас.

Каркас AngularJS по-прежнему существует (теперь уже под названием Angular 1) и будет поддерживаться в ближайшие годы, но вслед за ним появился каркас Angular 2 — совсем другое инструментальное средство для написания клиентских приложений в перспективе. Отказавшись от целого ряда антишаблонов, разработчики оформили Angular 2 в изящное и аккуратное программное средство, предвидя грядущее возрождение веб-технологий, опираясь, в частности, на стандарт ES6 языка JavaScript, веб-компоненты, рабочие веб-процессы, а также возможности реактивного программирования на языке TypeScript. Все это позволило поднять модульность каркаса Angular 2 на новые высоты и построить его по такому принципу, чтобы любую его модульную часть можно было бы отвергнуть или заменить. А самое главное, что Angular 2 предлагает обширный набор инструментальных средств для написания и конфигурирования приложений, способных работать с головокружительной быстротой.

Angular 2 может отпугнуть многих разработчиков, поскольку в нем немало нового и незнакомого. В данной книге читателю предлагается удобный путь к освоению и применению Angular 2 с наибольшей пользой. На простых примерах читатель сможет усвоить самые основы, а более сложные примеры раскрывают истинный потенциал данного каркаса. Эта книга организована в виде сборника независимых рецептов, что дает возможность приступить к изучению Angular 2 с любого места.

О чем эта книга?

В этой книге излагается материал, соответствующий версии 2.4 и вплоть до версии 4.0 каркаса Angular. А в приведенных примерах отсутствует исходный код из бета-версии или предвыпускных версий.

Я!

В главе 1, "Стратегии перехода к Angular 2", дается краткий обзор различных путей переноса приложений из Angular 1 в Angular 2. И хотя для данной цели не существует единой универсальной стратегии, в рецептах этой главы демонстрируются некоторые способы, позволяющие сохранить немалую часть уже существующей кодовой базы Angular 1.

В главе 2, "Знакомство с компонентами и директивами", приводится обширный ряд примеров, демонстрирующих назначение и применение компонентов Angular 2. Клиентские приложения создаются исключительно на компонентах Angular 2, и в этой главе полностью раскрывается их роль.

В главе 3, "Создание реактивных и управляемых по шаблонам форм", рассматриваются модули форм, полностью переработанные в Angular 2. Выстраивать свойства форм в Angular 2 можно двумя основными способами, подробно поясняемыми в этой главе.

В главе 4, "Использование обязательств", показана роль объекта типа `Promise` в Angular 2. И хотя в библиотеке RxJS реализованы некоторые полезные свойства обязательств, они относятся к объектам первого класса в стандарте ES6 языка JavaScript, которым по-прежнему принадлежит решающая роль.

В главе 5, "Наблюдаемые объекты из библиотеки ReactiveX", дается краткий курс по реактивному программированию в Angular 2. В него входят рецепты, демонстрирующие основы применения наблюдаемых объектов типа `Observable` и субъектов типа `Subject`, а также перспективные реализации, в которых предельно используются возможности библиотеки RxJS.

В главе 6, "Модуль Component Router", рассматривается полностью переработанный модуль маршрутизации в Angular 2. Здесь даются основы маршрутизации, а также поясняется целый ряд развитых концепций, характерных для Angular 2.

В главе 7, "Службы, внедрение зависимостей и класс NgModule", описываются новые и усовершенствованные в Angular 2 стратегии внедрения зависимостей. Здесь поясняется все, что требуется знать, чтобы разбивать приложения на независимые службы и модули, а также представлены стратегии, идеально подходящие для составления всех этих частей в единое целое.

В главе 8, "Организация приложений и управление ими", подробно поясняется, как управлять приложением Angular 2 на стороне клиента и за ее пределами. В каркасе Angular 2 внедрен целый ряд уровней сложности, требующих развитых инструментальных средств, и в этой главе даются советы по их применению.

В главе 9, "Тестирование в Angular 2", поясняется, как организуются тестовые наборы для Angular 2 и как пишутся различные виды тестов для этих наборов. Изучая новый каркас, многие разработчики избегают тестирования, и в этой главе ненавязчиво даются наставления по организации превосходной среды тестирования в Angular 2.

В главе 10, "Производительность и развитые концепции", дается краткий курс по целому ряду сложных понятий, внедренных в Angular 2. Здесь рассматриваются вопросы организации и архитектуры программного обеспечения, функциональные возможности и инструментальные средства данного каркаса, а также способы оптимизации во время компиляции.

Что требуется для этой книги

Каждый рецепт в этой книге снабжен ссылкой на сопутствующий ей вебсайт по адресу `http://ngcookbook.herokuapp.com/`. Если рецепты содержат примеры исходного кода, они снабжены дополнительными ссылками на общедоступный для разработчиков ресурс Plunker. Это дает возможность анализировать и тестировать исходный код в реальном времени, не прибегая к его компиляции и не требуя локальных серверов или чего-нибудь другого в том же роде. Следует, однако, иметь в виду, что такой подход пригоден лишь для экспериментирования, но не для разработки приложений, нацеленных на пользователей и эксплуатацию в реальных условиях.

Имеются две разновидности Angular 2 на JavaScript и TypeScript, но в этой книге рассматривается вариант на TypeScript, поскольку он превосходен синтаксически, как станет ясно в дальнейшем. Для разработки готовых к эксплуатации приложений исходный код на TypeScript должен быть преобразован в JavaScript-код прежде, чем его можно будет запустить в браузере. Для целей данной книги (и решения многих других задач подготовки исходного кода) это делается на платформе Node.js, установленной на локальной машине. В состав Node.js входит диспетчер пакетов (Node Package Manager — npm), позволяющий устанавливать и выполнять программное обеспечение на JavaScript с открытым кодом в режиме командной строки.

Для проработки материала некоторых глав придется установить платформу Node.js, чтобы выполнять команды и тестовые наборы на локальном сервере. Рекомендуется также, хотя это и не обязательно, установить на платформе Node.js диспетчер версий (Node Version Manager), упрощающий управление установленными пакетами.

Кому адресована эта книга

В настоящее время обширные материалы по изучению Angular 2 весьма разрознены. Эта книга адресована начинающим разработчикам, стремящимся освоить новый каркас или библиотеку, а также опытным разработчикам, интересующимся расширением своих знаний каркаса, поддерживающего перспективные веб-техноогии.

Усвоить сразу все эти новые технологии начинающим разработчикам будет нелегко. Поэтому материал данной книги организован таким образом,

чтобы постепенно раскрывать представленные в ней темы, а также поясняемые проектные решения и подробные обоснования.

Условные обозначения, принятые в книге

Различные виды информации в тексте книги представлены отдельными стилями. Ниже приведены некоторые примеры применения подобных стилей и поясняется их назначение.

Элементы исходного кода, имена таблиц базы данных, папок, файлов, их расширений и путей к ним, фиктивных URL, маркеров Twitter, вводимых пользователем данных представлены в тексте следующим образом: "Karma читает свою конфигурацию из файла `karma.conf.js`".

Блок исходного кода представлен таким образом:

```
<p>{{date}}</p>
<h1>{{title}}</h1>
<h3>Written by: {{author}}</h3>
```

Если требуется привлечь внимание к отдельной части блока кода, соответствующие его строки или элементы выделяются полужирным шрифтом, как показано ниже.

```
@Component({
  selector: 'article',
  template: `
    <p>{{currentDate|date}}</p>
    <h1>{{title}}</h1>
    <h3>Written by: {{author}}</h3>
  `
})
```

Любая информация, вводимая или выводимая в командной строке, представлена в следующем виде:

```
npm install karma jasmine-core karma-jasmine --save-dev
npm install karma-cli -g
```

Новые термины и важные понятия выделяются *курсивом*. Приведенные ниже пиктограммы обозначают следующее.

Предупреждения или важные примечания.

Советы и рекомендуемые приемы.

Загрузка исходного кода примеров

Исходные файлы примеров из этой книги можно бесплатно загрузить из информационного хранилища GitHub по адресу `https://github.com/PacktPublishing/Angular-2-Cookbook`. А другие комплекты исходного кода из обширного каталога книг и видеоматериалов издательства Packt Publishing Ltd. можно найти по адресу `https://github.com/PacktPublishing/`. Кроме того, вы можете загрузить архивный файл с исходными кодами примеров с веб-страницы книги на сайте издательской группы "Диалектика-Вильямс" по адресу `http://www.williamspublishing.com/Books/978-5-9909446-6-4.html`.

Как только архивный файл будет загружен, распакуйте его в отдельную папку, используя последнюю версию одной из следующих программ:

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

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш веб-сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

E-mail: `info@williamspublishing.com`
WWW: `http://www.williamspublishing.com`

Глава 1

Стратегии перехода к Angular 2

В этой главе даются следующие рецепты.

- Разбиение директив на компоненты путем инкапсуляции в свойстве `controllerAs`
- Перенос приложения в директивы компонентов
- Реализация основного компонента в версии AngularJS 1.5
- Нормализация типов служб
- Связывание Angular 1 и Angular 2 средствами класса `UpgradeModule`
- Обратный переход от компонентов Angular 2 к директивам Angular 1 с помощью функции `downgradeComponent()`
- Обратный переход от поставщиков Angular 2 к службам Angular 1 с помощью функции `downgradeInjectable()`

Введение

Внедрение версии Angular 2 в экосистему Angular, безусловно, будет воспринято разработчиками по-разному. Одни из них будут и далее придерживаться своих кодовых баз Angular 1, другие начнут с совершенно новых кодовых баз Angular 2, а третьи выберут постепенный переход.

Прежде чем перейти к представленным здесь рецептам, рекомендуется ознакомиться с поведением компонентов Angular 2. Это поможет вам установить свои мысленные модели в определенные рамки по мере адаптации существующих приложений к стилю, в большей степени соответствующему Angular 2.

Разбиение директив на компоненты путем инкапсуляции в свойстве `controllerAs`

К числу необычных условий, внедренных в Angular 1, относится взаимосвязь директив и данных, которые они потребляют. По умолчанию в директивах применялась наследуемая область видимости, вполне удовлетворявшая потребностям большинства разработчиков. Несмотря на всю простоту использования, такое условие вносило дополнительные зависимости в директивы, а также определяло другое условие, по которому директивы нередко не владели теми данными, которые они применяли. Кроме того, было неясно, чему именно присваивались и принадлежали данные, интерполированные в шаблоне.

Компоненты применяются в Angular 2 в качестве стандартных блоков всего приложения. Эти компоненты основаны на классах, а следовательно, в некоторых случаях они противоречат механизмам областей видимости в Angular 1. Поэтому переход от ориентированной на контроллеры модели директив является крупным шагом в направлении соответствия нормам Angular 2.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/8194/**.

Подготовка

Допустим, приложение содержит следующий код настройки, включающий в себя вложенные директивы, разделяющие общие данные в изолированной области видимости:

```
[index.html]

<div ng-app="articleApp">
  <article></article>
</div>
[app.js]

angular.module('articleApp', [])
.directive('article', function() {
  return {
    controller: function($scope) {
       $scope.articleData = {
          person: {firstName: 'Jake'},
          title: 'Lesotho Yacht Club Membership Booms'
       };
    },
     template: `
```

```
      <h1>{{articleData.title}}</h1>
      <attribution author="articleData.person.firstName">
      </attribution>
    `
  };
})
.directive('attribution', function() {
    return {
        scope: {author: '='},
        template: `<p>Written by: {{author}}</p>`
    };
});
```

Реализация

Цель состоит в том, чтобы реорганизовать приведенный выше код настройки, явно указав в шаблонах источник данных и сделав директивы владельцами этих данных, как выделено ниже полужирным шрифтом.

[app.js]

```
angular.module('articleApp', [])
.directive('article', function() {
  return {
    controller: function() {
      this.person = {firstName: 'Jake'};
      this.title = 'Lesotho Yacht Club Membership Booms';
    },
      controllerAs: 'articleCtrl',
      template: `
        <h1>{{articleCtrl.title}}</h1>
        <attribution></attribution>
      `
  };
})
.directive('attribution', function() {
  return {
    template: `<p>Written by: {{articleCtrl.author}}</p>`
  };
});
```

В приведенной ниже второй реализации рассматриваемого здесь рецепта можно с уверенностью определить источник данных статьи, где бы они ни использовались. И хотя эта реализация более совершенна, тем не менее, порожденная директива по-прежнему ссылается в ней на родительский контроллер, что нельзя назвать идеальным решением, поскольку оно вносит ненужную зависимость. Экземпляру директивы присваивания атрибутов

должны быть предоставлены данные, и вместо прежнего в нем должна быть выполнена интерполяция из его собственного экземпляра контроллера, как выделено ниже полужирным шрифтом.

[app.js]

```
angular.module('articleApp', [])
.directive('article', function() {
  return {
    controller: function() {
       this.person = {firstName: 'Jake'};
       this.title = 'Lesotho Yacht Club Membership Booms';
     },
      controllerAs: 'articleCtrl',
      template: `
        <h1>{{articleCtrl.title}}</h1>
        <attribution author="articleCtrl.person.firstName">
        </attribution>
      `
  };
})
.directive('attribution', function() {
  return {
    controller: function() {},
    controllerAs: 'attributionCtrl',
    bindToController: {author: '='},
    template: `<p>Written by: {{attributionCtrl.author}}</p>`
  };
});
```

Данная реализация выглядит намного лучше! В частности, порожденной директиве предоставляется подставляемый контроллер и задается псевдоним в шаблоне `attributionCtrl`. Она неявно привязывается к экземпляру контроллера через свойство `bindToController` таким же образом, как и в обычно изолированной области видимости. Но в данном случае привязка приписывается объекту контроллера, а не области видимости.

А теперь, когда введено понятие владения данными, допустим, что требуется модифицировать прикладные данные. Более того, их требуется модифицировать в разных частях приложения. Наивная реализация подобной задачи может выглядеть, например, следующим образом:

[app.js]

```
angular.module('articleApp', [])
.directive('attribution', function() {
  return {
    controller: function() {
      this.capitalize = function() {
```

```
        this.author = this.author.toUpperCase();
      }
    },
     controllerAs: 'attributionCtrl',
     bindToController: {author: '='},
     template: `
       <p ng-click="attributionCtrl.capitalize()">
         Written by: {{attributionCtrl.author}}
       </p>`
  };
});
```

В качестве желательного поведения пользователь может щелкнуть на имени автора, чтобы выделить его с прописной буквы. Но в приведенной выше реализации данные из контроллера статьи модифицируются в контроллере присваивания атрибутов, которыми он не владеет. Поэтому было бы лучше, чтобы данные модифицировал владеющий ими контроллер и вместо прежнего предоставлял интерфейс, которым мог бы воспользоваться внешний элемент (в данном случае — директива присваивания атрибутов):

```
[app.js]

angular.module('articleApp', [])
.directive('article', function() {
  return {
    controller: function() {
      this.person = {firstName: 'Jake'};
      this.title = 'Lesotho Yacht Club Membership Booms';
        this.capitalize = function() {
          this.person.firstName =
            this.person.firstName.toUpperCase();
        };
    },
     controllerAs: 'articleCtrl',
     template: `
       <h1>{{articleCtrl.title}}</h1>
       <attribution author="articleCtrl.person.firstName"
                    upper-case-author="articleCtrl.capitalize()">
       </attribution>
     `
  };
})
.directive('attribution', function() {
  return {
    controller: function() {},
    controllerAs: 'attributionCtrl',
    bindToController: {
      author: '=',
      upperCaseAuthor: '&'
```

```
    },
    template: `
      <p ng-click="attributionCtrl.upperCaseAuthor()">
        Written by: {{attributionCtrl.author}}
      </p>`
  };
});
```

Это уже весьма значительное усовершенствование! Оно по-прежнему допускает пространство имен в пределах привязки к щелчкам кнопками мыши. Но в то же время владеющий контроллер директив предоставляет внешним элементам метод, а не только непосредственный доступ к данным.

Принцип действия

Когда контроллер указывается в объекте определения директивы его экземпляр будет явным образом получен для каждого создаваемого экземпляра директивы. И вполне естественно, что этот объект контроллера будет инкапсулировать данные, которыми он владеет, поручая ответственность за передачу своих данных тем членам, которым они требуются.

В окончательной реализации приняты следующие меры.

- **Улучшение организации пространства имен.** Использование свойств объекта `$scope`, охватывающих несколько директив или вложения, создает ситуацию, когда несколько элементов могут манипулировать данными и читать их безотносительно к их источнику и контроллеру.
- **Улучшение тестируемости.** Если проанализировать директивы в окончательной реализации, то можно обнаружить, что протестировать их совсем не просто. У директивы присваивания атрибутов отсутствуют все зависимости, кроме явно передаваемой.
- **Инкапсуляция.** Введение в приложение понятия владения данными обеспечивает намного более надежную структуру, повышает степень использования кода, дает дополнительное представление и контроль над взаимодействием отдельных частей приложения.
- **Стиль Angular 2.** Аннотации `@Input` и `@Output` применяются в Angular 2 в определениях компонентов. Зеркальное отображение данного стиля способно заметно упростить процесс переноса приложения.

Дополнение

Следует заметить, что в приведенных выше примерах кода объект `$scope` был сделан совершенно неуместным. И хорошо, что в Angular 2 он вообще не упоминается, а это означает, что разработка приложения направлена на то, чтобы сделать его обновляемым. Нельзя сказать, что объект `$scope` совсем не

находит применения в приложениях Angular 1. В некоторых случаях вроде вызова метода `$scope.$apply()` без него просто не обойтись. Тем не менее представление об отдельных частях приложения как о компонентах позволяет лучше подготовиться к принятию условий Angular 2.

См. также

- Рецепт "Перенос приложения в директивы компонентов", демонстрирующий, как реорганизовать приложение Angular 1 в компонентном стиле.
- Рецепт "Реализация основного компонента в версии AngularJS 1.5", в котором подробно поясняется, как написать компонент Angular 1.
- Рецепт "Нормализация типов служб", в котором даются советы, как сделать типы служб Angular 1 совместимыми с Angular 2.

Перенос приложения в директивы компонентов

В каркасе Angular 1 имеется несколько встроенных директив, включая `ngController` и `ngInclude`, на которые разработчики обычно опираются при написании приложений. И хотя эти средства не являются антишаблонами, их применение уводит разработчиков в сторону от приложения, ориентированного на компоненты. Все эти директивы на самом деле являются подмножеством функциональных возможностей компонентов, и поэтому они могут быть полностью реорганизованы.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/1008/**.

Подготовка

Допустим, что первоначально приложение имеет следующий вид:

```
[index.html]

<div ng-app="articleApp">
  <ng-include src="'/press_header.html'"></ng-include>
  <div ng-controller="articleCtrl as article">
    <h1>{{article.title}}</h1>
    <p>Written by: {{article.author}}</p>
  </div>
  <script type="text/ng-template"
          id="/press_header.html">
  <div ng-controller="headerCtrl as header">
```

```
    <strong>
      Angular Chronicle - {{header.currentDate | date}}
    </strong>
    <hr />
  </div>
  </script>
</div>
[app.js]

angular.module('articleApp', [])
.controller('articleCtrl', function() {
  this.title = 'Food Fight Erupts During Diplomatic Luncheon';
  this.author = 'Jake';
})
.controller('headerCtrl', function() {
  this.currentDate = new Date();
});
```

В данном примере приложения содержится немало весьма распространенных в Angular 1 шаблонов. В частности, директивы **ngController** рассеяны по всему приложению, а в заголовочной части применяется директива **ngInclude**. Однако эти директивы непригодны для правильно сформированного приложения Angular 1. Но это положение можно исправить, реорганизовав исходный код по методике компонентно-ориентированного проектирования.

Реализация

Компонентно-ориентированные шаблоны не должны отпугивать своим внешним видом. В рассматриваемом здесь примере (а по существу, во всех приложениях Angular 1) можно реорганизовать компоненты, вообще не затрагивая существующий шаблон.

Начните с директивы `ngInclude`, которую нетрудно преобразовать в директиву компонента. В итоге она станет директивой с путем к шаблону, установленным в свойстве `templateUrl`:

```
[index.html]

<div ng-app="articleApp">
  <header></header>
  <div ng-controller="articleCtrl as article">
    <h1>{{article.title}}</h1>
    <p>Written by: {{article.author}}</p>
  </div>
  <script type="text/ng-template"
          id="/press_header.html">
  <div ng-controller="headerCtrl as header">
     <strong>
```

```
      Angular Chronicle - {{header.currentDate | date}}
    </strong>
    <hr />
  </div>
  </script>
</div>
[app.js]

angular.module('articleApp', [])
.controller('articleCtrl', function() {
  this.title = 'Food Fight Erupts During Diplomatic Luncheon';
  this.author = 'Jake';
})
.controller('headerCtrl', function() {
  this.currentDate = new Date();
})
.directive('header', function() {
  return {
    templateUrl: '/press_header.html'
  };
});
```

Далее можно реорганизовать директиву ngController везде, где она появляется в приложении. В данном примере она имеется в двух весьма характерных местах. Во-первых, она присутствует в заголовке шаблона press_header.html, действуя в качестве контроллера верхнего уровня в данном шаблоне. Зачастую это приводит к необходимости использовать излишний элемент оболочки лишь для того, чтобы разместить в нем шаблон. И во-вторых, директива ngController вложена в шаблон первичного приложения и управляет какой-нибудь произвольной частью модели DOM. В обоих этих местах можно выполнить преобразование в директивы компонентов, переназначив директиву ngController для контроллера директивы следующим образом:

```
[index.html]

<div ng-app="articleApp">
  <header></header>
  <article></article>
</div>
[app.js]

angular.module('articleApp', [])
.directive('header', function() {
  return {
    controller: function() {
      this.currentDate = new Date();
    },
```

```
    controllerAs: 'header',
    template: `
      <strong>
        Angular Chronicle - {{header.currentDate | date}}
      </strong>
      <hr />
    `
  };
})
.directive('article', function() {
  return {
    controller: function() {
      this.title = 'Food Fight Erupts During Diplomatic Luncheon';
      this.author = 'Jake';
    },
    controllerAs: 'article',
    template: `
      <h1>{{article.title}}</h1>
      <p>Written by: {{article.author}}</p>
    `
  };
});
```

В данном примере шаблоны включены в директиву ради наглядности и согласованности. А в крупных приложениях рекомендуется пользоваться свойством **templateUrl**, вынося разметку шаблона в отдельный файл.

Принцип действия

Вообще говоря, приложение можно представить в виде иерархии вложенных компонентов проектного шаблона MVC. Директивы `ngInclude` и `ngController` действуют в качестве подмножества функциональных возможностей компонента, поэтому их целесообразно расширить до полноценных директив компонента.

В рассматриваемом здесь примере окончательная структура приложения состоит только из компонентов, где каждому компоненту делегируется свой шаблон, контроллер и модель (через объект самого контроллера). Приверженцы скрупулезной точности могут оспорить, действительно ли каркас Angular реализован точно по шаблону MVC, но в контексте реорганизации компонентов это не так важно. В данном случае определена полностью модульная, повторно используемая, тестируемая, абстрагируемая и легко сопровождаемая структура. И сделано это в стиле Angular 2, который можно сразу же оценить по достоинству.

Дополнение

Внимательный разработчик заметит, что в данном рецепте не уделено никакого внимания наследованию области видимости. Это трудноразрешимый вопрос, главным образом, потому, что многие шаблоны в Angular 1 предназначены для преодоления несоответствий между областью видимости и свойством `controllerAs`. Каркас Angular 2 построен на строгом соответствии входных и выходных данных. А в Angular 1 область видимости наследуется по умолчанию, и поэтому объекты контроллеров, объемлющие вложенные директивы, доступны в них по умолчанию.

Следовательно, чтобы действительно смоделировать стиль Angular 2, придется настроить приложение таким образом, чтобы явно передавать данные и методы порожденным компонентам подобно тому, как это делается в рецепте "Разбиение директив на компоненты путем инкапсуляции в свойстве `controllerAs`". Но в то же время это никак не препятствует прямому доступу к контроллерам директив родительских компонентов. Впрочем, делать этого не следует, чтобы не вносить дополнительные зависимости.

См. также

Рецепт "Разбиение директив на компоненты путем инкапсуляции в свойстве `controllerAs`", в котором показан более совершенный способ организации директив Angular 1.

- Рецепт "Реализация основного компонента в версии AngularJS 1.5" с подробным пояснением, как написать компонент Angular 1.
- Рецепт "Нормализация типов служб", в котором даются советы, как сделать типы служб Angular 1 совместимыми с Angular 2.

Реализация основного компонента в версии AngularJS 1.5

В версии AngularJS 1.5 был внедрен компонент в качестве нового инструментального средства. И хотя он неточно соответствует понятию компонента в Angular 2, тем не менее, позволяет строить отдельные части приложения, составленные в директивном стиле, явно разбивая их на компоненты.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **`http://ngcookbook.herokuapp.com/7756/`**.

Подготовка

Допустим, в приложении имеется директива, определенная следующим образом:

```
[index.html]

<div ng-app="articleApp">
  <article></article>
</div>
[app.js]

angular.module('articleApp', [])
.directive('article', function() {
  return {
    controller: function() {
      this.person = {firstName: 'Jake'};
      this.title = 'Police Bust Illegal Snail Racing Ring';
      this.capitalize = function() {
        this.person.firstName =
        this.person.firstName.toUpperCase();
      };
    },
     controllerAs: 'articleCtrl',
     template: `
       <h1>{{articleCtrl.title}}</h1>
       <attribution author="articleCtrl.person.firstName"
                    upper-case-author="articleCtrl.capitalize()">
       </attribution>
     `
  };
})
.directive('attribution', function() {
  return {
    controller: function() {},
    controllerAs: 'attributionCtrl',
    bindToController: {
      author: '=',
      upperCaseAuthor: '&'
    },
     template: `
       <p ng-click="attributionCtrl.upperCaseAuthor()">
          Written by: {{attributionCtrl.author}}
       </p>`
  };
});
```

Реализация

Данное приложение уже организовано на основе инкапсуляции в свойстве `controllerAs`, поэтому его можно перенести в версию Angular 1.5, чтобы воспользоваться внедренным в ней определением функции `component()`.

Компоненты принимают определение объекта аналогично директиве, но функция совсем не обязательно должна возвращать объект — ей достаточно возвратить объектный литерал. В этом определении объекта свойство bindings употребляется таким же образом, как и свойство bindToController в директивах, что дает возможность без особого труда внедрить в данное приложение компоненты вместо директив:

```
[index.html]

<div ng-app="articleApp">
  <article></article>
</div>
[app.js]

angular.module('articleApp', [])
.component('article', {
  controller: function() {
    this.person = {firstName: 'Jake'};
    this.title = ' Police Bust Illegal Snail Racing Ring ';
    this.capitalize = function() {
      this.person.firstName =
         this.person.firstName.toUpperCase();
    };
  },
   controllerAs: 'articleCtrl',
   template: `
     <h1>{{articleCtrl.title}}</h1>
     <attribution author="articleCtrl.person.firstName"
                  upper-case-author="articleCtrl.capitalize()">
     </attribution>`
  })
  .component('attribution', {
    controller: function() {},
    controllerAs: 'attributionCtrl',
    bindings: {
      author: '=',
      upperCaseAuthor: '&'
    },
     template: `
     <p ng-click="attributionCtrl.upperCaseAuthor()">
                Written by: {{attributionCtrl.author}}
     </p>
    `
});
```

Принцип действия

Обратите внимание на то, что организация данных вокруг контроллера соответствует тому, что предполагается в определении компонента, и поэтому

никаких модификаций в шаблоне не требуется. По умолчанию в компонентах употребляется изолированная область видимости. Более того, объекты, окружающие компоненты, недоступны их псевдониму, чего нельзя сказать о директивах, написанных в компонентном стиле. Подобная инкапсуляция служит важным дополнением нового средства компонента, поскольку она прямо соответствует принципу действия компонентов в Angular 2.

Дополнение

В связи с тем что каждый компонент полностью изолирован, каждым шаблоном теперь оперирует отдельный объект контроллера. Таким образом, в версии Angular 1.5 автоматически предоставляется удобный псевдоним `$ctrl` для объекта контроллера отдельного компонента. И это делается независимо от того, указан или нет псевдоним `controllerAs`. Следовательно, в результате дальнейшей реорганизации получается следующий исходный код:

```
[index.html]

<div ng-app="articleApp">
  <article></article>
</div>
[app.js]

angular.module('articleApp', [])
.component('article', {
  controller: function() {
    this.person = {firstName: 'Jake'};
    this.title = 'Police Bust Illegal Snail Racing Ring';
    this.capitalize = function() {
      this.person.firstName =
        this.person.firstName.toUpperCase();
    };
  },
   template: `
     <h1>{{$ctrl.title}}</h1>
     <attribution author="$ctrl.person.firstName"
                  upper-case-author="$ctrl.capitalize()">
     </attribution>
   `
})
.component('attribution', {
  controller: function() {},
  bindings: {
    author: '=',
    upperCaseAuthor: '&'
  },
   template: `
   <p ng-click="$ctrl.upperCaseAuthor()">
```

```
        Written by: {{$ctrl.author}}
    </p>
    `
});
```

См. также.

- Рецепт "Разбиение директив на компоненты путем инкапсуляции в свойстве `controllerAs`", описывающий более совершенный способ организации директив Angular 1.
- Рецепт "Перенос приложения в директивы компонентов", в котором демонстрируется, как реорганизовать приложение Angular 1 в компонентном стиле.
- Рецепт "Нормализация типов служб", подсказывающий, как сделать типы служб Angular 1 совместимыми с Angular 2.

Нормализация типов служб

Разработчикам приложений Angular 1 должна быть знакома триада фабрика/служба/поставщик. Она не претерпела особых изменений в Angular 2. Но в интересах обновления и упрощения переноса существующего приложения необходимо исключить фабрики, заменив их службами.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/5637/**.

Подготовка

Допустим, имеется следующее простое приложение:

```
[index.html]

<div ng-app="articleApp">
<article></article>
</div>
[app.js]

angular.module('articleApp', [])
.factory('ArticleData', function() {
  var title = 'Incumbent Senator Ousted by Stalk of Broccoli';
    return {
      getTitle: function() {
        return title;
      },
       author: 'Jake'
```

```
    };
})
.component('article', {
  controller: function(ArticleData) {
    this.title = ArticleData.getTitle();
    this.author = ArticleData.author;
  },
   template: `
     <h1>{{$ctrl.title}}</h1>
     <p>Written by: {{$ctrl.author}}</p>
   `
});
```

Реализация

Каркас Angular 2 основывается на классах и включает в себя типы служб. В рассматриваемом здесь примере отсутствует тип службы, совместимый со структурой класса, а следовательно, он требует дополнительного преобразования. Правда, сделать это совсем не трудно, как выделено ниже полужирным шрифтом.

```
[index.html]

<div ng-app="articleApp">
  <article></article>
</div>
[app.js]

angular.module('articleApp', [])
.service('ArticleData', function() {
  var title = 'Incumbent Senator Ousted by Stalk of Broccoli';
  this.getTitle = function() {
    return title;
  };
  this.author = 'Jake';
})
.component('article', {
  controller: function(ArticleData) {
    this.title = ArticleData.getTitle();
    this.author = ArticleData.author;
  },
   template: `
     <h1>{{$ctrl.title}}</h1>
     <p>Written by: {{$ctrl.author}}</p>
   `
});
```

Принцип действия

В данном случае требуется не только сделать свойство `title` закрытым, но и сохранить интерфейс API, предоставляемый внедренным типом службы. Службы определяются в функции, действующей в качестве конструктора, и в конечном итоге внедряется экземпляр, создаваемый этим конструктором. С этой целью метод `getTitle()` и свойство `author`, определяемые по ссылкам `this`, перемещаются в отдельную службу. И благодаря этому данное свойство становится общим для всех экземпляров. Однако применение этих средств в компоненте и шаблоне ничего не меняет, да и не должно ничего менять.

Дополнение

Чтобы упростить реализацию типов служб, следует иметь в виду, что в Angular 1 фабрики зачастую применялись в первую очередь многими разработчиками, включая и самого автора книги. Некоторые разработчики могут не согласиться с тем, что для одновременного существования фабрик и служб нет никаких оснований. И те и другие обладают высокой степенью избыточности, но если углубиться в анализ исходного кода Angular, то можно обнаружить, что они, по существу, сводятся к одним и тем же методам.

Почему же службам следует отдавать предпочтение над фабриками? Новый стандарт ES6 языка JavaScript и язык TypeScript основываются на классах, поскольку это намного более изящный способ выражения и организации логики. Службы в Angular 1 реализованы на основе прототипных классов, которые, по существу, действуют аналогично формальным классам в версии языка JavaScript и TypeScript, если они употребляются правильно. Если остановиться на этом, то в конечном итоге получатся модифицированные службы, допускающие дальнейшее расширение и универсализацию. А если потребуется обновление, то окажется, что службы Angular 1 аккуратно обновятся до служб Angular 2.

См. также

- Рецепт "Разбиение директив на компоненты путем инкапсуляции в свойстве `controllerAs`", описывающий более совершенный способ организации директив Angular 1.
- Рецепт "Перенос приложения в директивы компонентов", в котором показано, как реорганизовать приложение Angular 1 в компонентном стиле.
- Рецепт "Реализация основного компонента в версии AngularJS 1.5", подробно поясняющий, как написать компонент Angular 1.

Связывание Angular 1 и Angular 2 средствами класса `UpgradeModule`

Компонент Angular 2 можно связать с существующим приложением Angular 1. Это свойство можно считать очевидным преимуществом, поскольку оно позволяет совместно использовать существующие компоненты и службы из Angular 1 и Angular 2. Для этой цели в Angular предоставляется инструментальное средство в виде класса `UpgradeModule`.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/4137/**.

Подготовка

Допустим, имеется следующее очень простое приложение Angular 1:

```
[index.html]

<!DOCTYPE html>
<html>
  <head>
    <!-- Сценарии в Angular 1 -->
    <script src="angular.js"></script>
  </head>
  <body>
    <div ng-app="hybridApp"
         ng-init="val='Angular 1 bootstrapped successfully!'">
      {{val}}
    </div>
  </body>
</html>
```

В этом приложении интерполируется значение, устанавливаемое в выражении Angular. Это дает возможность наглядно убедиться в том, что приложение запущено и нормально работает.

Реализация

Прежде всего объявите в собственном файле модуль `angular` верхнего уровня. Вместо того чтобы извлекать этот модуль с помощью дескриптора `script`, импортируйте каркас Angular 1 и создайте корневой модуль Angular 1, как показано ниже.

```
[ng1.module.ts]

import 'angular'

export const Ng1AppModule = angular.module('Ng1AppModule', []);
```

В состав Angular 2 входит готовый модуль обновления, доступный в файле `upgrade.js`. Оба упоминаемых здесь каркаса можно связать вместе средствами класса `UpgradeModule`.

В данном рецепте используется загрузчик модулей SystemJS и язык TypeScript со спецификациями в очень сложном файле конфигурации, который поясняется в следующей главе, поэтому не обращайте особого внимания на конкретные подробности реализации. А до тех пор необходимо принять во внимание, что загрузчик модулей SystemJS:

- настраивается на оперативную компиляцию исходных файлов TypeScript (с расширением **`.ts`**);
- способен выполнять операторы **`import`** и **`export`** в исходных файлах TypeScript;
- способен выполнять операции импорта библиотек в Angular 1 и 2.

Для каркаса Angular 2 требуется следующее определение модуля верхнего уровня как часть его базовой конфигурации:

```
[app/ng2.module.ts]

import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {UpgradeModule} from '@angular/upgrade/static';
import {RootComponent} from './root.component';

@NgModule({
  imports: [
    BrowserModule,
    UpgradeModule,
  ],
  bootstrap: [
    RootComponent
  ],
  declarations: [
     RootComponent
  ]
})
export class Ng2AppModule {
  constructor(public upgrade: UpgradeModule){}
}
export class AppModule {}
```

Причина, по которой выбрано именно такое определение модуля, не имеет особого значения для уяснения данного рецепта. Более подробно модули Angular 2 рассматриваются в главе 7, "Службы, внедрение зависимостей и класс NgModule".

Далее создайте корневой компонент приложения Angular 2, как показано ниже.

```
[app/root.component.ts]
import {Component} from '@angular/core';

@Component({
  selector: 'root',
  template: `
    <p>Angular 2 bootstrapped successfully!</p>
  `
})
export class RootComponent {}
```

Зачастую модуль Angular 2 запускается на выполнение из файла верхнего уровня, поэтому создайте данный файл под именем `main.ts` и запустите модуль Angular 2, как показано ниже.

```
[main.ts]
import {platformBrowserDynamic}
        from '@angular/platform-browser-dynamic';

import {Ng1AppModule} from './app/ng1.module';
import {Ng2AppModule} from './app/ng2.module';

platformBrowserDynamic().bootstrapModule(Ng2AppModule);
```

Связывание Angular 1 и Angular 2

Вместо директивы `ng-app` для запуска приложения Angular 1 после запуска модуля Angular 2 сделайте следующее:

```
[main.ts]

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

import {Ng1AppModule} from './app/ng1.module';
import {Ng2AppModule} from './app/ng2.module';

platformBrowserDynamic()
  .bootstrapModule(Ng2AppModule)
  .then(ref => {
    ref.instance.upgrade
       .bootstrap(document.body, [Ng1AppModule.name]);
  });
```

Благодаря этому у вас появится возможность удалить сценарий Angular 1 на JavaScript и директиву ng-app, а добавить корневой элемент приложения Angular 2:

```
[index.html]

<!DOCTYPE html>
<html>
  <head>
    <!-- Сценарии в Angular 2 -->
      <script src="zone.js "></script>
      <script src="reflect-metadata.js"></script>
      <script src="system.js"></script>
      <script src="system-config.js"></script>
  </head>
  <body>
    <div ng-init="val='Angular 1 bootstrapped successfully!'">
      {{val}}
    </div>
    <root></root>
  </body>
</html>
```

В приведенных выше новых сценариях наблюдаются зависимости от приложения Angular 2, но для уяснения данного рецепта понимать характер этих зависимостей совсем не обязательно. Более подробно они поясняются далее в книге.

Если выполнить все описанные выше действия, то шаблон приложения Angular 1 будет скомпилирован, а компонент Angular 2 отобразится надлежащим образом. Это, по существу, означает, что каркасы Angular 1 и Angular 2 благополучно действуют вместе.

Принцип действия

Чтобы не допустить ошибку, применяя класс UpgradeModule, следует создать приложения Angular 1 и Angular 2 на одной и той же странице, связав их вместе. Этот экземпляр адаптера позволит связать отдельные компоненты каждого из рассматриваемых здесь каркасов, чтобы гармонично пользоваться ими. В частности, на верхнем уровне создается приложение Angular 1 и появляется возможность использовать в нем отдельные компоненты Angular 2.

Дополнение

Несмотря на то что данный рецепт удобен для экспериментирования и обновления, он непригоден для любого приложения, которое должно действовать в реальном приложении. Ведь в этом случае удваивается величина

информационной поддержки со стороны обоих каркасов и вносится дополнительная сложность в существующее приложение. И хотя каркас Angular 2 намного более производительный, не следует ожидать одинаково приличных результатов от взаимообогащения приложений средствами `UpgradeModule`. Как следует из этого и представленных далее рецептов, компонентами Angular 2 можно пользоваться в приложениях Angular 1, используя методы трансляции адаптеров.

См. также

- Рецепт "Обратный переход от компонентов Angular 2 к директивам Angular 1 с помощью функции `downgradeComponent()`", в котором показано, как пользоваться компонентами Angular 2 в приложениях Angular 1.
- Рецепт "Обратный переход от поставщиков Angular 2 к службам Angular 1 с помощью функции `downgradeInjectable()`", в котором показано, как пользоваться службами Angular 2 в приложениях Angular 1.

Обратный переход от компонентов Angular 2 к директивам Angular 1 с помощью функции `downgradeComponent()`

Если вы внимательно следовали рецепту "Связывание Angular 1 и Angular 2 средствами класса `UpgradeModule`", то у вас должно получиться гибридное приложение, способное использовать различные элементы совместно с противоположным каркасом.

Если вы незнакомы с компонентами Angular 2, проработайте сначала материал следующей главы, посвященной компонентам.

Этот рецепт позволит в полной мере воспользоваться компонентами Angular 2 в шаблоне Angular 1.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **`http://ngcookbook.herokuapp.com/1499/`**.

Подготовка

Допустим, имеется следующий компонент Angular 2, которым требуется воспользоваться в приложении Angular 1:

```
[app/article.component.ts]

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

@Component({
  selector: 'ng2-article',
  template: `
    <h1>{{title}}</h1>
    <p>Written by: {{author}}</p>
  `
})
export class ArticleComponent {
  @Input() author:string
  title:string = 'Unicycle Jousting Recognized as Olympic Sport';
}
```

Итак, начните с реализации рецепта "Связывание Angular 1 и Angular 2 средствами класса UpgradeModule".

Реализация

В каркасе Angular 1 отсутствует какое-либо представление о том, как применять компоненты Angular 2. Существующий каркас Angular 2 добросовестно отобразит модуль, если представится такая возможность, но само определение модуля должно быть связано с каркасом Angular 1, чтобы его можно было запрашивать по мере надобности.

Сначала добавьте объявления компонентов в определение модуля, как выделено ниже полужирным шрифтом. Оно послужит для связывания обоих каркасов.

```
[app/app.module.ts]

import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {UpgradeModule} from '@angular/upgrade/static';
import {RootComponent} from './root.component';
import {ArticleComponent} from './article.component';

@NgModule({
  imports: [
    BrowserModule,
    UpgradeModule,
  ],
  declarations: [
    RootComponent,
    ArticleComponent
  ],
  bootstrap: [
    RootComponent
```

```
  ]
})
export class Ng2AppModule {
  constructor(public upgrade: UpgradeModule){}
}
```

Подобным образом объявление компонента связывается с контекстом Angular 2, хотя в Angular 1 по-прежнему отсутствует представление о сопряжении с ним. В таком случае придется воспользоваться функцией `downgradeComponent()`, чтобы определить компонент Angular 2 в качестве директивы Angular 1. Итак, назначьте для директивы Angular 1 другой дескриптор HTML-разметки, чтобы убедиться, что визуализация в нем выполняется средствами Angular 1, а не Angular 2:

```
[main.ts]

import {Component, Input} from '@angular/core';
import {downgradeComponent} from '@angular/upgrade/static';
import {Ng1AppModule} from './ng1.module';

@Component({
  selector: 'ng2-article',
  template: `
    <h1>{{title}}</h1>
    <p>Written by: {{author}}</p>
  `
})
export class ArticleComponent {
  @Input() author:string
  title:string = 'Unicycle Jousting Recognized as Olympic Sport';
}

Ng1AppModule.directive(
            'ng1Article',
            downgradeComponent({component: ArticleComponent}));
```

И, наконец, входное значение для компонента необходимо передать через атрибут привязки. Воспользуйтесь для этой цели синтаксисом привязки в Angular 2, как выделено ниже полужирным шрифтом, несмотря на то, что компонент по-прежнему объявляется в качестве директивы Angular 1.

```
[index.html]

<!DOCTYPE html>
<html>
  <head>
    <!-- Сценарии в Angular 2 -->
    <script src="zone.js "></script>
    <script src="reflect-metadata.js"></script>
```

```
    <script src="system.js"></script>
    <script src="system-config.js"></script>
  </head>
  <body>
    <div ng-init="authorName='Jake Hsu'">
      <ng1-article [author]="authorName"></ng1-article>
    </div>
    <root></root>
  </body>
</html>
```

Входные и выходные данные должны быть объявлены явным образом во время преобразования:

```
[app/article.component.ts]

import {Component, Input} from '@angular/core';
import {downgradeComponent} from '@angular/upgrade/static';
import {Ng1AppModule} from './ng1.module';

@Component({
  selector: 'ng2-article',
  template: `
    <h1>{{title}}</h1>
    <p>Written by: {{author}}</p>
  `
})
export class ArticleComponent {
  @Input() author:string
  title:string = 'Unicycle Jousting Recognized as Olympic Sport';
}

Ng1AppModule.directive(
  'ng1Article',
  downgradeComponent({
    component: ArticleComponent,
    inputs: ['author']
}));
```

Вот, собственно, и все требующиеся действия. Выполнив их надлежащим образом, вы должны увидеть, что в компоненте отображается имя автора, интерполируемое в компоненте Angular 2 через определение директивы `ng-init` в Angular 1.

Принцип действия

В данном рецепте каркасу Angular 1 предоставляется возможность направить каркас Angular 2 к определенному элементу модели DOM, дав ему команду на отображение в указанном месте. Каркас Angular 2 по-прежнему

управляет представлением и действием компонента, поэтому самое главное — полностью приспособить компонент Angular 2 к применению в шаблоне Angular 1.

Функции **`downgradeComponent()`** передается в качестве аргумента объект, определяющий компонент, а она возвращает функцию, которую каркас Angular 1 ожидает получить в качестве определения директивы.

См. также

- Рецепт "Связывание Angular 1 и Angular 2 средствами класса `UpgradeModule`", описывающий, как совместно применять каркасы Angular 1 и Angular 2.
- Рецепт "Обратный переход от поставщиков Angular 2 к службам Angular 1 с помощью функции `downgradeInjectable()`", в котором демонстрируется, как пользоваться службами Angular 2 в приложениях Angular 1.

Обратный переход от поставщиков Angular 2 к службам Angular 1 с помощью функции `downgradeInjectable()`

Если вы внимательно следовали рецепту "Связывание Angular 1 и Angular 2 средствами класса `UpgradeModule`", то у вас должно получиться гибридное приложение, способное использовать различные элементы совместно с противоположным каркасом. А если вы незнакомы с поставщиками (providers) в Angular 2, то рекомендуется проработать сначала материал главы 7, посвященной внедрению зависимостей.

Аналогично шаблонным компонентам, взаимозаменяемость предоставляется и для типов служб. В частности, тип службы можно сначала определить в Angular 2, а затем внедрить его в контексте Angular 1.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **`http://ngcookbook.herokuapp.com/2824/`**.

Подготовка

Начните с примера кода, написанного в рецепте "Связывание Angular 1 и Angular 2 средствами класса `UpgradeModule`".

Реализация

Определите сначала службу, которую требуется внедрить в компонент Angular 1, как показано ниже.

```
[app/article.service.ts]

import {Injectable} from '@angular/core';

@Injectable()
export class ArticleService {
  article:Object = {
    title: 'Research Shows Moon Not Actually Made of Cheese',
    author: 'Jake Hsu'
  };
}
```

Далее определите компонент Angular 1, который требуется внедрить в дальнейшем:

```
[app/article.component.ts]

export const ng1Article = {
  template: `
    <h1>{{article.title}}</h1>
    <p>{{article.author}}</p>
  `,
  controller: (ArticleService, $scope) => {
    $scope.article = ArticleService.article;
  }
};
```

Службу `ArticleService` пока еще нельзя внедрить, поскольку в каркасе Angular 1 отсутствует всякое представление о существовании службы. Впрочем, сделать это нетрудно. Прежде всего укажите поставщика службы в определении модуля Angular 2, как это обычно делается:

```
[app/ng2.module.ts]

import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {UpgradeModule} from '@angular/upgrade/static';
import {RootComponent} from './root.component';
import {ArticleService} from './article.service';

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

```
  declarations: [
    RootComponent
  ],
  providers: [
    ArticleService
  ],
  bootstrap: [
    RootComponent
  ]
})
export class Ng2AppModule {
  constructor(public upgrade: UpgradeModule){}
}
```

Тем не менее в каркасе Angular 1 по-прежнему неизвестно, как пользоваться службой. Аналогичным образом, определение компонента Angular 2 преобразуется в директиву Angular 1, а служба Angular 2 — в фабрику Angular 1. Воспользуйтесь далее функцией `downgradeInjectable()`, чтобы добавить компонент Angular 1 и преобразованную службу в определение модуля Angular 1, как показано ниже.

`[app/ng1.module.ts]`

```
import 'angular';
import {ng1Article} from './article.component';
import {ArticleService} from './article.service';
import {downgradeInjectable} from '@angular/upgrade/static';

export const Ng1AppModule = angular.module('Ng1AppModule', [])
  .component('ng1Article', ng1Article)
  .factory('ArticleService', downgradeInjectable(ArticleService));
```

Вот и все! В итоге в компоненте Angular 1 должны быть отображены данные, переданные из службы Angular 2.

См. также

- Рецепт "Связывание Angular 1 и Angular 2 средствами класса `UpgradeModule`", в котором демонстрируется, как совместно применять каркасы Angular 1 и Angular 2.
- Рецепт "Обратный переход от компонентов Angular 2 к директивам Angular 1 с помощью функции `downgradeComponent()`", описывающий, как пользоваться компонентами Angular 2 в приложениях Angular 1.

Глава 2

Знакомство с компонентами и директивами

В этой главе даются рецепты на тот случай, если требуется выполнить итерацию по содержимому страницы и отобразить название статьи только, если она активизирована.

- Применение декораторов для создания и стилизации простого компонента
- Передача членов родительского компонента порожденному компоненту
- Привязка к естественным атрибутам элементов разметки
- Регистрация обработчиков естественных событий в браузерах
- Генерирование и обработка специальных событий средствами класса `EventEmitter`
- Присоединение поведения к элементам модели DOM с помощью директив
- Отображение вложенного содержимого с помощью директивы `ngContent`
- Применение структурных директив `ngFor` и `ngIf` для управления на основе модели DOM
- Обращение к элементам разметки с помощью переменных шаблона
- Привязка свойств атрибутов
- Применение перехватчиков жизненного цикла компонентов
- Обращение к родительскому компоненту из порожденного компонента

- Настройка взаимного оповещения родительских и порожденных компонентов с помощью декоратора `@ViewChild` и функции `forwardRef()`
- Настройка взаимного оповещения родительских и порожденных компонентов с помощью декоратора `@ContentChild` и функции `forwardRef()`

Введение

С директивами, знакомыми вам по Angular 1, теперь покончено. Вместо них в Angular 2 появились два новых элемента: компоненты и новая версия директив. Теперь приложения Angular 2 ориентированы на компоненты, и в дальнейшем станет ясно, почему компонентный стиль разработки приложений оказывается более предпочтительным.

Большая часть синтаксиса в Angular 2 полностью обновлена и на первый взгляд может показаться необычной. Но не бойтесь! Изящность и чудесность основ компонентного стиля в Angular 2 станут очевидными, стоит только уяснить их как следует.

Применение декораторов для создания и стилизации простого компонента

Для написания компонента приложения на TypeScript необходимо уяснить и усвоить ряд новых парадигм. И хотя это поначалу может оказаться непростым делом, тем не менее, в процессе усвоения у вас появится возможность воспользоваться накопленным багажом знаний о директивах в Angular 1.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **`http://ngcookbook.herokuapp.com/6577/`**.

Подготовка

К числу самых простых компонентов, которые только можно создать, относится элемент шаблона, вставляющий некоторые значения в его шаблон. В каркасе Angular 1 этого можно, например, добиться, создав директиву элемента, включив некоторые данные в область видимости связывающей функции и присоединив шаблон для обращения к этим данным. Такое описание выбрано, в частности, потому, что практически все эти понятия уже устоялись.

Допустим, требуется создать простой компонент статьи с псевдошаблоном, как показано ниже. Этот компонент должен находиться в естественном дескрипторе HTML-разметки, визуализировать шаблон и подставлять значения.

```
<p>{{date}}</p>
<h1>{{title}}</h1>
<h3>Written by: {{author}}</h3>
```

Реализация

Элементарным стандартным блоком для написания приложений в Angular 2 служит компонент. В общем, этот компонент можно определить двумя средствами: определением базового класса и декорированием класса.

Написание определения базового класса

Все компоненты Angular 2 начинаются как классы. Такой класс служит для получения экземпляра компонента, и любые требующиеся в шаблоне компонента данные могут быть получены из свойств класса. Таким образом, базовый класс для компонента будет выглядеть следующим образом:

```
[app/article.component.ts]

export class ArticleComponent {
  currentDate:date;
  title:string;
  author:string;
  constructor() {
    this.currentDate = new Date();
    this.title = `
      Flight Security Restrictions to Include
      Insults, Mean Faces
    `;
    this.author = 'Jake';
  }
};
```

Начинающим изучать язык TypeScript или версию ES6 языка JavaScript необходимо иметь в виду следующее.

- Определение класса предваряется ключевым словом `export` в соответствии с новым условным обозначением модулей в стандарте ES6 языка JavaScript, которое является также естественной частью языка TypeScript. Так, если в исходном файле `foo.ts` определен класс `Article`, его можно импортировать в другой модуль, используя ключевое слово `import` и указав в нем путь к этому модулю следующим образом (здесь предполагается, что импортирующий файл находится в том же каталоге, где исходный файл `foo.ts`):

  ```
  import {Article} from './foo';
  ```

- При определении заголовка употребляется новый синтаксис символьных строк шаблона, принятый в стандарте ES6 языка JavaScript: пара

обратных кавычек (``) вместо традиционных ("). Это очень удобное обозначение вместо маловразумительной мешанины '' + '' + '', употреблявшейся ранее для определения многострочных шаблонов.

- Все свойства данного класса типизированы. Синтаксис типизации в языке TypeScript принимает следующую форму:

  имяСвойства:*типСвойства* = *необязательноеНачальноеЗначение*

  Язык JavaScript, безусловно, является слабо типизированным, и в данном случае именно его сценарии интерпретируются в браузере. Тем не менее разработка приложения на TypeScript позволяет использовать типовую безопасность во время компиляции и тем самым избежать нежелательного и непредсказуемого поведения.

- Во всех классах в стандарте ES6 языка JavaScript предопределен метод `constructor()`, который вызывается при создании экземпляра. В данном случае конструктор применяется для создания экземпляров свойств класса, что идеально подходит для инициализации его членов. Возможность определять свойство члена класса за пределами его конструктора оказывается удобной для снабжения свойств типами. Это позволяет просто отказаться от использования конструктора, поскольку в одной строке можно указать тип значения и тут же присвоить его. Таким образом, более лаконичный стиль программирования может выглядеть следующим образом:

```
[app/article.component.ts]

export class ArticleComponent {
  currentDate:date = new Date();
  title:string = `
    Flight Security Restrictions to Include
    Insults, Mean Faces
  `;
  author:string = 'Jake';
}
```

- Компилятор языка TypeScript автоматически перенесет процесс инициализации членов класса в его конструктор. Поэтому данная версия прикладного кода по своему поведению идентична предыдущей.

Написание декоратора класса компонента

Несмотря на то что ранее был создан класс вместе со связанной с ним информацией, в нем пока еще отсутствует какой-либо способ сопряжения с моделью DOM. Более того, этот класс еще предстоит снабдить каким-либо смыслом в контексте Angular 2. И это можно сделать с помощью декораторов.

Декораторы являются языковыми средствами TypeScript, но они ни в коем случае не характерны только для этого языка. В частности, программирующим на языке Python хорошо известен принцип модуляризации классов посредством явного декорирования. Вообще говоря, это дает возможность получить упорядоченную модификацию определенных классов, используя отдельно определенные декораторы. Но в Angular 2 предоставляются декораторы, применяемые, главным образом, для объявления различных элементов данного каркаса. В частности, декоратор **`@Component`** определяется в исходном коде Angular в виде функции **`Component()`**, которая применяется как декоратор с префиксом `@`.

Префикс `@` свидетельствует о том, что импортируемая функция должна применяться как декоратор. И хотя такие декораторы вполне наглядны, обычно они не существуют сами по себе или в виде пустого объектного литерала. Дело в том, что декораторы, как правило, приносят пользу в Angular 2 благодаря их *метаданным*. Это понятие можно продемонстрировать здесь на конкретном примере декоратора `Component`.

Но в Angular 2 ничего не дается бесплатно. Чтобы воспользоваться декоратором компонента в Angular 2, его придется импортировать из модуля `core` в тот модуль, где требуется им воспользоваться. С этой целью определение декоратора `@Component` можно поместить перед определением класса, как выделено ниже полужирным шрифтом.

```
[app/article.component.ts]

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

@Component({})

export class ArticleComponent {
  currentDate:date = new Date();
  title:string = `
    Flight Security Restrictions to Include
    Insults, Mean Faces
  `;
  author:string = 'Jake';
}
```

Как упоминалось выше, декоратору `@Component` метаданные передаются в виде объекта типа `ComponentMetadata`, который в приведенном выше примере кода представлен пустым объектным литералом (между прочим, этот код не будет компилироваться). Принципиально такой объект метаданных очень похож на объект, определяющий директиву в Angular 1. В данном случае требуется предоставить объект с метаданными декоратора и двумя свойствами `selector` и `template`, как показано ниже.

```
[app/article.component.ts]

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

@Component({
  selector: 'article',
  template: `
    <p>{{currentDate|date}}</p>
    <h1>{{title}}</h1>
    <h3>Written by: {{author}}</h3>
  `
})
export class ArticleComponent {
  currentDate:date = new Date();
  title:string = `
    Flight Security Restrictions to Include
    Insults, Mean Faces
  `;
  author:string = 'Jake';
}
```

Следует иметь в виду, что свойство `selector` содержит символьную строку, предназначенную для обозначения места, где компонент должен быть вставлен в модель DOM, а свойство `template`, очевидно, — строковое представление HTML-шаблона. Принимая все это во внимание, вы увидите, как действует компонент статьи, помещая контент внутрь дескрипторов `<article></article>`.

Принцип действия

Определение класса в Angular 1 было вытеснено благодаря введению концепции контроллера. Экземпляры компонентов данного класса содержат свойства-члены, которые могут быть вставлены и привязаны к шаблону аналогично объекту `$scope` в Angular 1.

В шаблоне процессы вставки и привязки данных должны, по-видимому, происходить во многом так же, как и в Angular 1. Но на самом деле это не так, как поясняется более подробно далее в этой главе. Встроенный модификатор `date`, напоминающий фильтр в Angular 1, теперь дублируется модификатором `pipe`, хотя он и действует очень похожим образом.

Свойство метаданных `selector` служит строковым представлением CSS-селектора. Данное определение влияет на все экземпляры дескриптора `article`, но особенности селектора таковы, что он может справиться и с более сложными ситуациями. Поэтому пользуйтесь этим свойством с наибольшей выгодой для себя.

Дополнение

Начинающие осваивать Angular 2 должны уяснить понятие полной инкапсуляции компонента. Различные возможности объекта типа `Component Metadata` подробнее рассматриваются далее в данной книге, но в этом и других классах декораторов вводится принцип *самоописания* компонентов в Angular 2. Анализируя определение класса, можно полностью осмыслить данные, службы, классы и внедряемые зависимости. А в Angular 1 это было невозможно из-за шаблона, порождающего "суп из областей видимости".

Можно, конечно, возразить, что в Angular 1 оформление CSS-стилями было второстепенным делом. Но благодаря компонентам это теперь уже не так, поскольку объект метаданных обеспечивает надежную поддержку сложного стилевого оформления. Например, чтобы выделить курсивом имя автора в компоненте `article`, можно воспользоваться следующим фрагментом кода:

```
[app/article.component.ts]

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

@Component({
  selector: 'article',
  template: `
    <p>{{currentDate|date}}</p>
    <h1>{{title}}</h1>
    <h3>Written by: {{author}}</h3>
  `,
  styles: [`
    h3 {
      font-style: italic;
    }
  `]
})
export class ArticleComponent {
  currentDate:date = new Date();
  title:string = `
    Flight Security Restrictions to Include
    Insults, Mean Faces
  `;
  author:string = 'Jake';
}
```

Приведенное выше свойство `styles` применяется в Angular 2 для компиляции в генерируемую таблицу стилей, применяемую только к данному компоненту. Не обращайте особого внимания на непреднамеренное стилевое оформление остальных HTML-дескрипторов `h3`. Оно объясняется тем, что генерируемая в Angular 2 стилевая таблица гарантирует, что только данный

компонент (и порожденные им компоненты) подчиняются правилам CSS в объекте метаданных.

Данный рецепт предназначен для эмулирования общей модульности веб-компонентов. Но поскольку веб-компоненты пока еще не нашли универсальной поддержки, то в архитектуре Angular 2, по существу, выполняется эмуляция (polyfill) данного поведения.

См. также

- Рецепт "Передача членов родительского компонента порожденному компоненту", поясняющий основы организации нисходящего потока данных между компонентами.
- Рецепт "Применение структурных директив `ngFor` и `ngIf` для управления на основе модели DOM", в котором даются советы о том, как использовать некоторые из базовых директив, встроенных в Angular 2.
- Рецепт "Применение перехватчиков жизненного цикла компонентов" с примером интеграции с потоком визуализации компонентов в Angular 2.

Передача членов родительского компонента порожденному компоненту

В связи с отступлением от принципа наследования объекта `$scope` в версии Angular 1.x приходится (полностью или частично) переосмысливать порядок передачи информации приложению. Вместо этого предоставляется совершенно новая система распространения информации по иерархии приложения.

Канул в Лету и принцип двунаправленной привязки данных по умолчанию. И хотя двунаправленная привязка данных упрощала осмысление приложения, она снижала производительность до непростительно низкого уровня. А новая система действует асимметрично, когда члены распространяются вниз, а не вверх по дереву компонентов, если только это не делается явно.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **`http://ngcookbook.herokuapp.com/6543/`**.

Подготовка

Допустим, имеется следующее простое приложение, предназначавшееся ранее (но не теперь) для передачи данных из родительского компонента `ArticleComponent` в порожденный компонент `AttributionComponent`:

[app/components.ts]

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

@Component({
  selector: 'attribution',
  template: `
    <h3>Written by: {{author}}</h3>
  `
})
export class AttributionComponent {
  author:string;
}
@Component({
  selector: 'article',
  template: `
    <h1>{{title}}</h1>
    <attribution></attribution>
  `
})
export class ArticleComponent {
  title:string = 'Belching Choir Begins World Tour';
  name:string = 'Jake';
}
```

Реализация

Компоненты, определенные в приведенной выше первоначальной реализации, пока еще не имеют ни малейшего представления друг о друге, а дескриптор <attribution></attribution> остается неактивным в модели DOM. И это хорошо, поскольку означает, что оба компонента совершенно разъединены, а следовательно, дает возможность внедрять логику их связывания по мере надобности.

Связывание компонентов

Прежде всего компоненту Article необходимо дать знать о существовании компонента AttributionComponent, поскольку дескриптор <attribution></attribution> находится в компоненте Article. С этой целью данный компонент внедряется в тот модуль, в котором объявлен компонент ArticleComponent:

[app/app.module.ts]

```
import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {ArticleComponent, AttributionComponent}
        from './components';
@NgModule({
```

```
  imports: [
    BrowserModule
  ],
  declarations: [
    ArticleComponent,
    AttributionComponent
  ],
  bootstrap: [
    ArticleComponent
  ]
})
export class AppModule {}
```

Таким образом, в компоненте `ArticleComponent` можно согласовать дескриптор `<attribution></attribution>` с определением компонента `AttributionComponent`.

Для понимания данного рецепта не обращайте пока что особого внимания на предназначение декоратора **@NgModule**. Все приложение в примере из данного рецепта является лишь экземпляром типа **ArticleComponent**, содержащим компонент типа **AttributionComponent**. Следодовательно, объявления всех компонентов могут быть сделаны в одном и том же модуле.

В единственном модуле порядок определения может иметь большое значение. А поскольку объявления классов в стандарте ES6 языка JavaScript и в языке TypeScript не поднимаются, то ссылаться на них прежде их объявления без возникновения ошибки нельзя. В данном рецепте компонент **ArticleComponent** определяется прежде компонента **Attribution Component**, и поэтому ссылаться непосредственно на второй из них в определении первого нельзя.

Если же, наоборот, определить компонент **AttributionComponent** в отдельном модуле и затем импортировать его с помощью загрузчика модулей, то порядок определения не имеет особого значения. Нетрудно заметить, что это одно из самых примечательных преимуществ, которое дает наличие модульной в высокой степени структуры приложения.

Следует, однако, оговориться, что в Angular нельзя ссылаться не по порядку на классы с помощью функции **forwardRef()**. Но если вопрос порядка определения можно разрешить, разделив приложение на отдельные модули, то такой способ следует предпочесть применению функции **forwardRef()**.

Исходя из этого, разбейте компонент в своем исходном файле на два отдельных модуля и соответственно импортируйте их, как показано ниже.

```
[app/app.module.ts]

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

```
import {BrowserModule} from '@angular/platform-browser';
import {ArticleComponent} from './article.component';
import {AttributionComponent} from './attribution.component';

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

[app/article.component.ts]

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

@Component({
  selector: 'article',
  template: `
    <h1>{{title}}</h1>
    <attribution></attribution>
    `
})
export class ArticleComponent {
  title:string = 'Belching Choir Begins World Tour';
  name:string = 'Jake';
}
```

[app/attribution.component.ts]

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

@Component({
  selector: 'attribution',
  template: `
    <h3>Written by: {{author}}</h3>
  `
})
export class AttributionComponent {
  author:string;
}
```

Объявление входных данных

Аналогично объекту свойств Angular 1 с областью видимости директивы, в Angular 2 необходимо объявлять члены родительского компонента, чтобы

они были видны в порожденном компоненте. В каркасе Angular 1 это можно было осуществить неявно с помощью наследуемого объекта `$scope`, но теперь уже так сделать нельзя. Входные данные компонента Angular 2 теперь должны быть определены явно.

Еще одно важное различие каркасов Angular 1 и Angular 2 заключается в том, что декоратор **`@Input`** в Angular 2 является средством однонаправленной привязки данных. Обновление данных распространяется вниз по иерархии, где родительский компонент не обновится, если он не будет уведомлен явным образом.

Процесс объявления входных данных в порожденном компоненте выполняется через декоратор `@Input`, но этот декоратор вызывается в самом определении класса, а не прежде него. Декоратор `@Input` импортируется из базового модуля и вызывается в определении класса вместе с его членом.

Но это не должно вводить вас в заблуждение. Реализация конкретной декорирующей функции скрывается от вас, потому что она импортируется как единый целевой объект. Поэтому не особенно задумывайтесь о назначении синтаксиса **`@Input()`**. В исходном коде Angular имеется определенная функция **`Input()`**, и в данном случае она, безусловно, вызывается как метод. Но в целях данного примера объявляется лишь следующий после нее член, который будет явным образом передаваться из родительского компонента. И это делается таким же образом, как и в декораторе **`@Component`**, но только в другом месте.

```
[app/attribution.component.ts]

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

@Component({
  selector: 'attribution',
  template: `
    <h3>Written by: {{author}} </h3>
  `
})
export class AttributionComponent {
  @Input() author:string;
}
```

Далее значение, привязанное к дескриптору порожденного компонента, необходимо передать родительскому компоненту. В контексте данного рецепта требуется передать свойство `name` объекта компонента типа `Article` свойству `author` объекта компонента типа `Attribution`. Этого можно, в частности, добиться с помощью обозначения в виде квадратных скобок в атрибуте дескриптора, где символьная строка определяется как привязанные данные,

как выделено ниже полужирным шрифтом. Таким образом, свойство члена успешно передано вниз по иерархии от родительского к порожденному компоненту!

```
[app/article.component.ts]

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

@Component({
  selector: 'article',
  template: `
    <h1>{{title}}</h1>
    <attribution [author]="name"></attribution>
  `
})
export class ArticleComponent {
  title:string = 'Belching Choir Begins World Tour';
  name:string = 'Jake';
}
```

Принцип действия

Напомним, что в самом начале рассматриваемого здесь примера два компонента не имели никакого представления друг о друге, несмотря на то, что они определяются и экспортируются в одном и том же модуле. Процесс, демонстрируемый в данном рецепте, служит для того, чтобы предоставить порожденный компонент родительскому компоненту, настроить порожденный компонент на ожидание привязки члена к атрибуту ввода данных и предоставить этот член в шаблоне родительского компонента.

Дополнение

Некоторые разработчики испытывают трудности, используя обозначение в виде квадратных скобок. Оно вполне допустимо в HTML, хотя далеко не все разработчики считают его интуитивным и удобным.

Следует также иметь в виду, что обозначение в виде квадратных скобок недопустимо в XML. Разработчики, пользующиеся HTML-разметкой, формируемой средствами XSLT, не смогут пользоваться этим новым синтаксисом. Правда, везде, где в синтаксисе Angular 2 применяются квадратные и круглые скобки, можно употребить эквивалентный синтаксис, поддерживаемый в данном каркасе с идентичным поведением.

Вместо употребления пар квадратных скобок имя атрибута можно предварить префиксом **bind-**, обеспечив идентичное поведение, как выделено ниже полужирным шрифтом.

[app/article.component.ts]

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

@Component({
  selector: 'article',
  template: `
    <h1>{{ title }}</h1>
    <attribution bind-author="name"></attribution>
  `
})
export class ArticleComponent {
  title:string = 'Belching Choir Begins World Tour';
  name:string = 'Jake';
}
```

Выражения в Angular

Следует иметь в виду, что значение атрибута `name` является не символьной строкой, а выражением. В каркасе Angular известно, как вычислить такое выражение в контексте родительского компонента. Но в таких выражениях все же рекомендуется указывать статические значения, чтобы каркас Angular благополучно вычислил их и предоставил порожденному компоненту.

Например, в следующем фрагменте жестко кодируется присваивание символьной строки `"Mike Snifferpippets"` свойству `author` порожденного компонента:

[app/article.component.ts]

```
import {Component, Input} from '@angular/core';
import {AttributionComponent} from './attribution.component';

@Component({
  selector: 'article',
  template: `
    <h1>{{title}}</h1>
    <attribution [author]="'Mike Snifferpippets'"></attribution>
  `,
  directives: [AttributionComponent]
})
export class ArticleComponent {
  title:string = 'Belching Choir Begins World Tour';
  name:string = 'Jake';
}
```

Однонаправленная привязка данных

Привязка данных, которую приходится устанавливать в данном рецепте, по существу, оказывается однонаправленной. В частности, изменения в члене

родительского компонента будут распространяться вниз по иерархии к порожденному компоненту, тогда как изменения в члене порожденного компонента не будут распространяться вверх по иерархии. Этот принцип будет дополнительно исследован в другом рецепте, но при этом следует иметь в виду, что потоки данных в Angular 2 по умолчанию распространяются вниз по иерархическому дереву компонентов.

Компонентные методы

Характер привязываемого значения не имеет особого значения в Angular. Правильность типов все равно будет соблюдена в языке TypeScript, даже если отклониться от объявленного типа. Тем не менее описываемую здесь стратегию рекомендуется применять и для передачи методов от родителей к потомкам.

Передача метода, привязываемого подобным способом, не обеспечивает контекст для его выполнения. Так, если родительский компонент передает компонентный метод, в котором применяется ключевое слово **this**, для последующего выполнения в порожденном компоненте, то по ссылке **this** произойдет обращение к экземпляру порожденного, а не родительского компонента. Следовательно, если в таком методе будет предпринята попытка получить доступ к данным в члене родительского компонента, они окажутся недоступными.

Данное затруднение можно разрешить целым рядом способов. Но, как правило, вместо передачи компонентного метода от родителя к потомку и последующего его вызова рекомендуется найти лучший способ разработки приложения.

См. также

- Рецепт "Применение декораторов для создания и стилизации простого компонента", в котором описываются стандартные блоки для реализации компонентов в Angular 2.
- Рецепт "Привязка к естественным атрибутам элементов разметки", демонстрирующий, каким образом Angular 2 сопрягается с атрибутами элементов HTML-разметки.
- Рецепт "Регистрация обработчиков естественных событий в браузерах", в котором показано, насколько просто присоединить поведение к событиям в браузере.
- Рецепт "Генерирование и обработка специальных событий средствами класса `EventEmitter`", подробно поясняющий, каким образом информация распространяется вверх по иерархии компонентов.

- Рецепт "Применение структурных директив ngFor и ngIf для управления на основе модели DOM" с советами о том, как использовать некоторые базовые директивы, встроенные в Angular 2.

Привязка к естественным атрибутам элементов разметки

В каркасе Angular 1 предполагалось, что разработчики должны использовать встроенные директивы замены для атрибутов элементов разметки, имевших присоединенное к ним значимое поведение модели DOM. Это объясняется тем, что многие из подобных атрибутов имели поведение, несовместимое с тем, как в Angular 1 выполнялась привязка данных. В каркасе Angular 2 эти специальные директивы атрибутов упразднены, а поведение и синтаксис привязки сводятся к обычному поведению привязки.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/8313/**.

Реализация

Чтобы осуществить привязку к естественному (native) атрибуту, достаточно заключить этот атрибут в квадратные скобки и интерпретировать ее как обычную привязку данных, как выделено ниже полужирным шрифтом. При такой настройке элемент разметки <img> добросовестно извлечет и отобразит изображение, когда оно предоставляется каркасом Angular.

```
[app/logo.component.ts]

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

@Component({
  selector: 'logo',
  template: '<img [src]="logoUrl">'
})
export class LogoComponent {
  logoUrl:string =
    '//angular.io/resources/images/logos/standard/logo-nav.png';
}
```

Принцип действия

Это совершенно другое решение той же самой проблемы, которая в Angular 1 решалась с помощью директивы ng-src. В частности, браузер ищет атрибут src в дескрипторе. Но поскольку квадратные скобки включаются как

часть в символьную строку атрибута, то браузер не обнаружит его, а следовательно, не сделает запрос. Обозначение `[src]` лишь приведет к запросу изображения, как только значение будет заполнено и предоставлено элементу разметки.

См. также

- Рецепт "Передача членов родительского компонента порожденному компоненту" с пояснениями основ организации нисходящего потока данных между компонентами.
- Рецепт "Регистрация обработчиков естественных событий в браузерах", в котором показано, насколько просто присоединить поведение к событиям в браузере.
- Рецепт "Присоединение поведения к элементам модели DOM с помощью директив", демонстрирующий, как присоединять поведение к элементам с помощью директив атрибутов.
- Рецепт "Обращение к элементам разметки с помощью переменных шаблона", в котором представлена новая конструкция переменных шаблона в Angular 2.
- Рецепт "Привязка свойств атрибутов" с описанием искусного способа углубленного обращения к свойствам элементов разметки в Angular 2.

Регистрация обработчиков естественных событий в браузерах

Другой разновидностью привязки в Angular 2, которая требуется для полного функционирования веб-приложения, является привязка событий. Синтаксис привязки событий в Angular 2 такой же, как и для привязки данных.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/4437/**.

Подготовка

Допустим, требуется создать приложение для подсчета количества загрузок отдельных статей. Начните его создание со следующей заготовки:

```
[app/article.component.ts]
import {Component} from '@angular/core';

@Component({
  selector: 'article',
```

```
  template: `
    <h1>{{title}}</h1>
    <p>Shares: {{shareCt}}</p>
    <button>Share</button>
  `
})
export class ArticleComponent {
  title:string = 'Police Apprehend Tiramisu Thieves';
  shareCt:number = 0;
}
```

Реализация

Синтаксис привязки событий в Angular 2 обозначается парой круглых скобок, в которые заключается тип события. В данном случае события, которые требуется принимать на обработку, будут иметь свойство типа `click`, к которому их предстоит привязать. В качестве значения атрибута привязки событий служит выражение, что дает возможность вызвать в нем метод их обработки:

[app/article.component.ts]

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

@Component({
  selector: 'article',
  template: `
    <h1>{{title}}</h1>
    <p>Shares: {{shareCt}}</p>
    <button (click)="share()">Share</button>
  `
})
export class ArticleComponent {
  title:string = 'Police Apprehend Tiramisu Thieves';
  shareCt:number = 0;
  share():void {
    ++this.shareCt;
  }
}
```

Принцип действия

Синтаксис привязки событий (`click`) отслеживается средствами Angular, и в конечном итоге в компонент `ArticleComponent` вводится приемник событий от щелчков кнопками мыши, привязанный к их обработчику `share()`. Когда наблюдается событие, вычисляется выражение, присоединяемое к данному событию. В данном случае вызывается метод, определенный в компоненте.

Дополнение

Обработка события должна происходить в выражении, поэтому в выражении предоставляется параметр `$event`, который обычно передается в качестве аргумента методу обработки событий. Этот процесс происходит таким же образом, как и в Angular 1. Анализируя объект в виде параметра `$event`, можно обнаружить в нем обыкновенное событие, сгенерированное браузером:

```
[app/article.component.ts]

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

@Component({
  selector: 'article',
  template: `
    <h1>{{title}}</h1>
    <p>Shares: {{shareCt}}</p>
    <button (click)="share($event)">Share</button>
  `
})
export class ArticleComponent {
  title:string = 'Police Apprehend Tiramisu Thieves';
  shareCt:number = 0;
  share(e:Event):void {
    console.log(e); // событие типа MouseEvent
    ++this.shareCt;
  }
}
```

Обратите также внимание, каким образом в методе **share()** демонстрируется применение типизации к параметрам и возврат значения. Ниже приведена общая форма объявления такого метода.

```
myMethod(arg1:arg1type, arg2:arg2type, ...):returnType
```

Аналогично привязке членов, для привязки событий можно применять альтернативный синтаксис, если неудобно пользоваться круглыми скобками. Предварив имя атрибута префиксом **on-**, можно добиться идентичного поведения, как выделено ниже полужирным шрифтом.

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

@Component({
  selector: 'article',
  template: `
    <h1>{{title}}</h1>
    <p>Shares: {{shareCt}}</p>
    <button on-click="share($event)">Share</button>
  `
})
```

```
export class Article {
  title:string = 'Police Apprehend Tiramisu Thieves';
  shareCt:number = 0;
  share(e:Event):void {
    ++this.shareCt;
  }
}
```

См. также

- Рецепт "Привязка к естественным атрибутам элементов разметки", демонстрирующий, каким образом Angular 2 сопрягается с атрибутами элементов HTML-разметки.
- Рецепт "Генерирование и обработка специальных событий средствами класса `EventEmitter`", в котором подробно поясняется, каким образом информация распространяется вверх по иерархии компонентов.
- Рецепт "Присоединение поведения к элементам модели DOM с помощью директив", описывающий, как присоединять поведение к элементам с помощью директив атрибутов.
- Рецепт "Привязка свойств атрибутов", в котором представлен искусный способ углубленного обращения к свойствам элементов разметки в Angular 2.

Генерирование и обработка специальных событий средствами класса `EventEmitter`

В связи с упразднением объекта `$scope` в Angular образовалось пустое место в механизме распространения информации вверх по иерархическому дереву компонентов. И это пустое место было отчасти заполнено специальными событиями, которые представляют собой *инь* (т.е. мужское начало) для нисходящей привязки данных к *янь* (т.е. к женскому началу).

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **`http://ngcookbook.herokuapp.com/8611/`**.

Подготовка

Допустим, имеется следующее приложение `Article`:

```
[app/text-editor.component.ts]

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

```
@Component({
  selector: 'text-editor',
  template: `
    <textarea></textarea>
  `
})
export class TextEditorComponent {}
```

```
[app/article.component.ts]

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

@Component({
  selector: 'article',
  template: `
    <h1>{{title}}</h1>
    <p>Word count: {{wordCount}}</p>
    <text-editor></text-editor>
  `
})
export class ArticleComponent {
  title:string = `
    Maternity Ward Resorts to Rock Paper Scissors Following
    Baby Mixup`;
  wordCount:number = 0;

  updateWordCount(e:number):void {
    this.wordCount = e;
  }
}
```

Данное приложение идеально подходит для чтения содержимого элемента разметки `textarea`, когда происходит изменение, а также для подсчета количества слов и его передачи родительскому компоненту для последующей вставки. Но в данном случае ничего из этого не реализовано.

Реализация

Разработчик, мыслящий категориями Angular 1, присоединит директиву `ng-model` к элементу разметки `textarea`, вызовет метод `$scope.$watch()` для данных модели и передаст данные родителю через объект `$scope` или другими средствами. К сожалению, для такого разработчика подобные конструкции коренным образом отличаются или вообще отсутствуют в Angular 2. Впрочем, отчаиваться не стоит! Новая реализация Angular оказывается более выразительной, модульной и ясной.

Перехват данных события

Директива `ngModel` по-прежнему существует в Angular 2 и, безусловно, ее можно использовать и в данном случае. Но теперь совсем не обязательно

пользоваться директивой ngModel, поскольку действие, предпринимаемое в приложении, можно выразить более явным образом. Прежде всего необходимо извлечь текст из элемента разметки textarea и сделать его пригодным для применения в компоненте TextEditorComponent:

```
[app/text-editor.component.ts]

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

@Component({
  selector: 'text-editor',
  template: `
  <textarea (keyup)="emitWordCount($event)"></textarea>
  `
})
export class TextEditorComponent {
  emitWordCount(e:Event):void {
    console.log(
      (e.target.value.match(/\S+/g) || []).length);
  }
}
```

Отлично! Как утверждалось выше, для получения содержимого элемента разметки совсем не обязательно пользоваться директивой ngModel. Более того, теперь можно применить естественные события в браузере, чтобы явно определить момент, когда требуется предпринять действие в компоненте TextEditorComponent.

Подобным образом устанавливается приемник естественных событий keyup в браузере, инициируемых в элементе разметки textarea. У данного события имеется свойство target, раскрывающее текстовое значение в элементе разметки. Именно оно здесь и используется. Затем для подсчета количества символов, следующих без пробелов, в данном компоненте применяется простое регулярное выражение. Это и есть подсчет слов.

Распространение специального события

Метод console.log() никак не помогает в уведомлении родительского компонента о подсчитанном количестве слов. Для этого придется сформировать специальное событие и распространить его вверх по иерархии, как показано ниже.

```
[app/text-editor.component.ts]

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

@Component({
  selector: 'text-editor',
```

```
  template: `
    <textarea (keyup)="emitWordCount($event)"></textarea>
  `
})
export class TextEditorComponent {
  @Output() countUpdate = new EventEmitter<number>();

  emitWordCount(e:Event):void {
    this.countUpdate.emit(
      (e.target.value.match(/\S+/g) || []).length);
  }
}
```

Применение декоратора `@Output` позволяет получить в порожденном компоненте отправитель событий как экземпляр члена класса `EventEmitter`, который способен принять родительский компонент. Как и любой другой член класса, этот член доступен по ссылке `this.countUpdate`. Порожденный компонент способен посылать события вверх по иерархии, вызывая метод `emit()` для данного члена, а в качестве аргумента этого метода задается значение, которое требуется послать событию. И поскольку здесь требуется послать целочисленный подсчет слов, необходимо получить экземпляр типа `EventEmitter`, задав его как отправитель событий типа `<number>`.

Прием специальных событий

Итак, половина реализации осуществлена, поскольку специальные события генерируются в среде браузера, но без их приемников. Требующийся для этого метод уже определен в родительском компоненте, поэтому остается лишь привязать к нему приемник событий.

Для привязки приемников к событиям служит синтаксис шаблонов `()`, а в Angular не проводится никаких отличий между естественными событиями в браузере и теми событиями, которые инициируются членами типа `EventEmitter`. А поскольку отправитель событий типа `EventEmitter` объявлен в порожденном компоненте как декоратор `@Output`, можно ввести приемник событий, поступающих из порожденного компонента в родительский, как показано ниже. Таким образом, приложение должно правильно подсчитывать слова в компоненте `TextEditor` и обновлять значение в компоненте `Article`.

[app/article.component.ts]

```
import {Component } from 'angular2/core';

@Component({
  selector: 'article',
  template: `
    <h1>{{title}}</h1>
```

```
    <p>Word count: {{wordCount}}</p>
    <text-editor (countUpdate)="updateWordCount($event)">
    </text-editor>
  `
})
export class ArticleComponent {
  title:string = `
    Maternity Ward Resorts to Rock Paper Scissors Following
    Baby Mixup`;
  wordCount:number = 0;

  updateWordCount(e:number):void {
  this.wordCount = e;
}
}
```

Принцип действия

Применяя декоратор `@Output` с классом `EventEmitter`, можно создавать порожденные компоненты, делающие доступным интерфейс API для привязки родительского компонента. Отправитель событий типа `EventEmitter` посылает события вверх по иерархии с помощью метода `emit()`, а родительский компонент может подписаться на эти события, привязавшись к выходным данным отправителя событий.

Ниже приведен порядок выполнения действий в рассматриваемом здесь примере.

1. В результате нажатия клавиши в элементе разметки `textarea` наступает естественное событие `keyup` в браузере.
2. Компонент `TextEditor` имеет установленный приемник для этого события, поэтому будет вычислено присоединенное выражение, где вызывается метод `emitWordCount()`.
3. В методе `emitWordCount()` проверяется объект типа `Event` и далее извлекается текст из связанного с ним элемента модели DOM. Полученный текст анализируется на количество содержащихся в нем слов и затем вызывается метод `EventEmitter.emit()`.
4. Метод `EventEmitter.emit()` распространяет событие, связанное с объявленным членом `countUpdate @Output`.
5. Компонент `ArticleComponent` обнаруживает данное событие и вызывает присоединенное выражение. А в этом выражении вызывается метод `updateWordCount()`, которому передается значение события.
6. Свойство компонента `ArticleComponent` обновляется, а поскольку его значение вставляется в представление, то Angular выполняет процесс привязки данных и обновляет это представление.

Дополнение

Название класса `EventEmitter` несколько обманчиво. Если присмотреться внимательнее, то можно заметить, что у метода родительского компонента, вызываемого в обработчике событий, отсутствует типизированный параметр. Можно также заметить, что этот параметр присваивается непосредственно члену типа `number`. И это может показаться странным, поскольку при вызове метода в шаблонном выражении передается параметр `$event`, использовавшийся ранее как объект типа `Event` события в браузере. В этом усматривается некоторое несоответствие. Ведь если привязать естественные события в браузере, то наблюдаемым может быть только объект естественного события в браузере. А если привязать специальные события, то наблюдаемым может быть любое событие, передаваемое при вызове метода `emit()`. Здесь же в качестве параметра метода `updateWordCount()` служит целочисленное значение, передаваемое при вызове метода `this.countUpdate.emit()`.

Следует также иметь в виду, что предоставлять значение для распространяемого события необязательно. Классом `EventEmitter` можно по-прежнему пользоваться для уведомления родительского компонента о том, что наступило событие и что в этом компоненте следует вычислить привязанное выражение. Для этого достаточно создать нетипизированный отправитель событий с помощью конструкции `new EventEmitter()` и вызвать метод `emit()` без аргументов. Параметр `$event` должен быть неопределенным (`undefined`).

Передать несколько значений в качестве специальных событий нельзя. Чтобы отправить несколько фрагментов данных, их придется объединить в объекте или массиве.

См. также

- Рецепт "Привязка к естественным атрибутам элементов разметки", демонстрирующий, каким образом Angular 2 сопрягается с атрибутами элементов HTML-разметки.
- Рецепт "Регистрация обработчиков естественных событий в браузерах", в котором показано, насколько просто присоединить поведение к событиям в браузере.

Присоединение поведения к элементам модели DOM с помощью директив

В процессе разработки приложений нередко оказывается полезным присоединить поведение в компонентном стиле к элементам модели DOM, не прибегая к шаблонной обработке. Но если попытаться создать компонент Angular 2, не предоставляя так или иначе шаблон, то появится суровое

сообщение об ошибке, извещающее о необходимости предоставить шаблон в той или иной форме.

Компоненты Angular 2 отличаются от директив тем, что у них имеются представления, принимающие форму свойства `template`, `templateUrl` или декоратора `@View`, тогда как у директив ничего этого нет. А в остальном они ведут себя одинаково, обеспечивая то же самое поведение.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/3292/**.

Подготовка

Допустим, имеется следующее приложение:

```
[app/article.component.ts]

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

@Component({
  selector: 'article',
  template: `<h1>{{title}}</h1>`,
  styles: [`
    h1 {
      text-overflow: ellipsis;
      white-space: nowrap;
      overflow: hidden;
      max-width: 300px;
    }
  `]
})
export class ArticleComponent {
  title:string = `Presidential Candidates Respond to
        Allegations Involving Ability to Dunk`;
}
```

В настоящий момент CSS используется в этом приложении с целью укоротить название статьи с помощью многоточия. Это приложение следовало бы расширить таким образом, чтобы компонент `Article` отображал название статьи полностью, просто добавляя атрибут HTML-разметки, если щелкнуть на нем.

Реализация

Начните с определения следующего основного класса, изменяющего директиву атрибута и вводящего ее в модуль приложения:

```
[app/click-to-reveal.directive.ts]

export class ClickToRevealDirective {
```

```
  reveal(target) {
    target.style['white-space'] = 'normal';
  }
}
```

[app/app.module.ts]

```
import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {ArticleComponent} from './article.component';
import {ClickToRevealDirective}
       from './click-to-reveal.directive';

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

Прежде всего необходимо декорировать класс `ClickToRevealDirective` в виде `@Directive` и воспользоваться им в компоненте `Article` следующим образом:

[app/click-to-reveal.directive.ts]

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

@Directive({
  selector: '[click-to-reveal]'
})
export class ClickToRevealDirective {
  reveal(target) {
    target.style['white-space'] = 'normal';
  }
}
```

Далее добавьте в элемент разметки атрибут, к которому требуется применить директиву:

[app/article.component.ts]

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

```
@Component({
  selector: 'article',
  template: `<h1 click-to-reveal>{{ title }}</h1>`,
  styles: [`
    h1 {
      text-overflow: ellipsis;
      white-space: nowrap;
      overflow: hidden;
      max-width: 300px;
    }
  `]
})
export class ArticleComponent {
  title: string;
  constructor() {
    this.title = `Presidential Candidates Respond to
    Allegations Involving Ability to Dunk`;
  }
}
```

CSS-селектор атрибутов применяется в декораторе **@Directive** для связывания с любыми элементами разметками, имеющими директиву **click-to-reveal**. Это, безусловно, напоминает поведение директивы атрибута в Angular 1, но данная форма оказывается намного более гибкой, поскольку позволяет овладеть совместимостью, внутренне присущей селекторам.

А теперь, когда компоненту `Article` стало известно о классе `ClickToRevealDirective`, необходимо предоставить ему возможность присоединяться к событиям от щелчков кнопками мыши.

Присоединение событий с помощью декораторов `@HostListener`

Внимательный читатель заметит, что до сих пор в этой главе речь шла о создании компонентов, обрабатывающих события, генерируемые порожденными компонентами. Это не вызывает особых затруднений, поскольку приемники событий могут быть выразительно установлены в шаблоне родительского компонента, задаваемого в порожденном дескрипторе.

Но в данном случае требуется найти способ ввести приемник событий в тот же самый элемент разметки, к которому присоединяется директива. Более того, подходящего способа для ввода выражения, привязывающего события, в шаблон из директивы не существует. В идеальном случае было бы желательно не раскрывать такой метод из директивы. Но как тогда поступить?

В качестве выхода из подобного затруднения можно воспользоваться новой в Angular конструкцией под названием `HostListener`. Проще говоря, эта конструкция позволяет перехватывать самонарождающиеся события и обрабатывать их внутренним образом, как показано ниже. Таким образом, при

наступлении событий от щелчков кнопками мыши (click) в элементе разметке <h1> должен вызываться метод reveal().

[app/click-to-reveal.directive.ts]

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

@Directive({
  selector: '[click-to-reveal]'
})
export class ClickToRevealDirective {
  @HostListener('click', ['$event.target'])
  reveal(target) {
    target.style['white-space'] = 'normal';
  }
}
```

Принцип действия

Директиве требуется какой-нибудь способ для присоединения естественных событий от щелчков кнопками мыши. Кроме того, ей требуется способ для обработки таких объектов, как $event, предоставляемых средствами Angular. Как правило, такие объекты обрабатываются в привязывающем выражении.

Декоратор @HostListener оформляет метод директивы в качестве назначенного обработчика событий. В качестве первого аргумента при вызове данного метода служит строка идентификации события (в данном случае — click, хотя это вполне могло быть и специальное событие из отправителя событий типа EventEmitter), а в качестве второго аргумента — массив строковых аргументов, вычисляемых как выражения.

Дополнение

В директиве можно указывать не только декоратор @HostListener. С его помощью можно лишь связать событие с методом директивы. Следовательно, для приема событий click и mouseover можно указать один за другим несколько объявлений декораторов @HostListener в одном обработчике событий.

См. также

- Рецепт "Применение декораторов для создания и стилизации простого компонента", описывающий стандартные блоки для реализации компонентов в Angular 2.
- Рецепт "Передача членов родительского компонента порожденному компоненту" с пояснениями основ организации нисходящего потока данных между компонентами.

- Рецепт "Применение структурных директив ngFor и ngIf для управления на основе модели DOM", в котором даются советы о том, как использовать некоторые из базовых директив, встроенных в Angular 2.
- Рецепт "Привязка свойств атрибутов", представляющий искусный способ углубленного обращения к свойствам элементов разметки в Angular 2.

Отображение вложенного содержимого с помощью директивы ngContent

Применение компонентов в качестве самостоятельных дескрипторов, полностью управляющих своим содержимым, является чистым шаблоном. Но нередко оказывается, что дескрипторы компонентов требуют, чтобы в них было заключено содержимое.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/6172/**.

Подготовка

Допустим, имеется следующее приложение:

```
[app/ad-section.component.ts]

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

@Component({
  selector: 'ad-section',
  template: `
    <a href="#">{{adText}}</a>
  `
})
export class AdSectionComponent {
  adText:string = 'Selfie sticks 40% off!';
}

[app/article.component.ts]

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

@Component({
  selector: 'article',
  template: `
    <h1>{{title}}</h1>
    <p>U.S. senators are up in arms following the recent ruling
       stripping them of their beloved selfie sticks.</p>
```

```
    <p>A bipartisan committee drafted a resolution to smuggle
       selfie sticks onto the floor by any means necessary.</p>
  `
})
export class ArticleComponent {
  title:string = 'Selfie Sticks Banned from Senate Floor';
}
```

Цель в данном случае состоит в том, чтобы модифицировать это приложение таким образом, чтобы внедрить компонент `AdSection` в компонент `Article`, не затрагивая его содержимое.

Реализация

Компонент `AdSection` требует внедрения дополнительного элемента вокруг содержимого, существующего в компоненте `Article`. Этого нетрудно добиться следующим образом:

```
[app/article.component.ts]

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

@Component({
  selector: 'article',
  template: `
    <h1>{{title}}</h1>
    <ad-section>
      <p>U.S. senators are up in arms following the recent ruling
         stripping them of their beloved selfie sticks.</p>
      <p>A bipartisan committee drafted a resolution to smuggle
         selfie sticks onto the floor by any means necessary.</p>
    </ad-section>
  `
})
export class ArticleComponent {
  title:string = 'Selfie Sticks Banned from Senate Floor';
}
```

Однако следует иметь в виду, что это деструктивная операция. При визуализации компонента `AdSectionComponent` в Angular во внимание не принимается находящееся в нем содержимое. При этом обнаруживается, что у компонента `AdSectionComponent` имеется связанный с ним шаблон, который благополучно подставляется вместо содержимого элемента разметки. И этот шаблон определяется в декораторе `@Component`. В данном случае это приводит к удалению дескрипторов `<p>`, которые требуется сохранить.

Чтобы сохранить их, необходимо проинструктировать Angular, как следует управлять содержимым, заключенным в оболочку. И этого можно добиться с помощью дескриптора `<ng-content>`, как показано ниже. Таким образом,

элемент привязки рекламы вводится прежде содержимого, заключаемого в оболочку.

```
[app/ad-section.component.ts]

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

@Component({
  selector: 'ad-section',
  template: `
    <a href="#">{{adText}}</a>
    <ng-content select="p"></ng-content>
  `
})
export class AdSectionComponent {
  adText:string = 'Selfie sticks 40% off!';
}
```

Принцип действия

Аналогично директиве `ng-transclude` в Angular 1, директива `ng-content` служит для вставки содержимого, заключаемого в оболочку дескриптора компонента, в его шаблон. Отличие здесь заключается в том, что атрибут `select` применяется в директиве `ng-content` с целью наметить элементы разметки, заключаемые в оболочку. Это просто CSS-селектор, действующий таким же образом, как и декораторы `@Component`, обрабатывающие свойство `selector` в объекте типа `ComponentMetadata`.

Дополнение

В данном примере атрибут `select` оказался излишним, поскольку он в конечном итоге выбирал все содержимое, заключенное в оболочку. Разумеется, если значение атрибута `select` совпадает только с некоторым содержимым, заключенным в оболочку, то извлечены и вставлены будут только соответствующие элементы разметки. Если же не предоставить значение атрибута `select` в дескрипторе `<ng-content>`, то по умолчанию будет введено все содержимое, заключенное в оболочку.

Следует также иметь в виду, что атрибут `select` является ограниченным CSS-селектором. Он не в состоянии выполнять сложные операции выделения вроде `:nth-child` и может только намечать элементы разметки верхнего уровня в дескрипторах заключения в оболочку. Например, в данном приложении отсутствует дескриптор абзацев, включаемый в разметку `<div><p>Blah</p></div>` со значением атрибута `select="p"`.

См. также

- Рецепт "Обращение к родительскому компоненту из порожденного компонента", описывающий, каким образом компонент может получить прямую ссылку на своего родителя через внедрение зависимостей.
- Рецепт "Настройка взаимного оповещения родительских и порожденных компонентов с помощью декоратора `@ViewChild` и функции `forwardRef()`", в котором даются советы о том, как следует пользоваться декоратором `@ViewChild` для ссылки на экземпляры объектов порожденных компонентов.
- Рецепт "Настройка взаимного оповещения родительских и порожденных компонентов с помощью декоратора `@ContentChild` и функции `forwardRef()`", в котором даются советы о том, как следует пользоваться декоратором `@ContentChild` и функцией `forwardRef()` для ссылки на экземпляры объектов порожденных компонентов.

Применение структурных директив `ngFor` и `ngIf` для управления на основе модели DOM

Любому разработчику, пользующемуся клиентским каркасом, хорошо знакомы две основные операции, применяемые в приложении: итеративная визуализация содержимого коллекции и условная визуализация. В новой реализации в Angular 2 они выглядят несколько иначе, хотя и действуют во многом таким же образом.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/3211/**.

Подготовка

Допустим, имеется приведенное ниже приложение. Цель в данном случае состоит в том, чтобы обойти коллекцию названий статей и отобразить их заголовок только в том случае, если они активны.

```
[app/article-list.component.ts]

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

@Component({
  selector: 'article-list',
  template: ''
})
export class ArticleListComponent {
```

```
  articles:Array<Object> = [
    {title: 'Foo', active: true},
    {title: 'Bar', active: false},
    {title: 'Baz', active: true}
  ];
}
```

Реализация

Как и в Angular 1, для решения подобной задачи в Angular 2 предоставляются соответствующие директивы. В частности, директива `ngFor` служит для перебора коллекции названий статей `articles`, как показано ниже.

```
[app/article-list.component.ts]

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

@Component({
  selector: 'article-list',
  template: `
    <div *ngFor="let article of articles; let i = index">
      <h1>
        {{i}}: {{article.title}}
      </h1>
    </div>
  `
})
export class ArticleListComponent {
  articles:Array<Object> = [
    { title: 'Foo', active: true },
    { title: 'Bar', active: false },
    { title: 'Baz', active: true }
  ];
}
```

Как и директиву `ngFor`, директиву `ngIf` можно внедрить так, как показано ниже. Как видите, в данном случае из массива `articles` визуализируются только объекты со значением свойства `active: true`.

```
[app/article-list.component.ts]

import {Component} from 'angular2/core';

@Component({
  selector: 'article-list',
  template: `
    <div *ngFor="let article of articles; let i = index">
      <h1 *ngIf="article.active">
        {{i}}: {{ article.title }}
```

```
      </h1>
    </div>
  `
})
export class ArticleListComponent {
  articles:Array<Object> = [
    {title: 'Foo', active: true},
    {title: 'Bar', active: false},
    {title: 'Baz', active: true}
  ];
}
```

Принцип действия

Прежде всего, обозначения директив знаками "звездочки" и "решетки" могут ввести в замешательство многих разработчиков. Для разработки многих приложений совсем не обязательно знать внутренний механизм работы этого синтаксического удобства.

В действительности все структурные директивы, предваряемые знаком *, подвергаются декомпозиции в Angular для употребления шаблона. Прежде всего, директивы `ngFor` и `ngIf` анализируются в Angular для применения директив `template` к одному и тому же элементу. Сам же синтаксис не претерпевает пока еще особых изменений:

```
[app/article-list.component.ts]

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

@Component({
  selector: 'article-list',
  template: `
    <div template="ngFor let article of articles; let i = index">
      <h1 template="ngIf article.active">
        {{i}}: {{ article.title }}
      </h1>
    </div>
  `
})
export class ArticleList {
  articles: Array<Object>,
  constructor() {
    this.articles = [
     { title: 'Foo', active: true },
     { title: 'Bar', active: false },
     { title: 'Baz', active: true }
    ];
  }
}
```

После этого директива template подвергается в Angular декомпозиции на заключающие в оболочку элементы разметки <template>, как показано ниже.

```
[app/article-list.component.ts]

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

@Component({
  selector: 'article-list',
  template: `
    <template ngFor let-article [ngForOf]="articles" let-i="index">
      <div>
        <template [ngIf]="article.active">
          <h1>
            {{i}}: {{article.title}}
          </h1>
        </template>
      </div>
    </template>
  `
})
export class ArticleList {
  articles: Array<Object>,
  constructor() {
    this.articles = [
      { title: 'Foo', active: true },
      { title: 'Bar', active: false },
      { title: 'Baz', active: true }
    ];
  }
}
```

Приложение в обеих приведенных выше версиях, где применяется директива **template** или элемент разметки **<template>**, ведет себя так же, как и в том случае, когда применяются первоначальные структурные директивы. В общем, поступать подобным образом нет никаких оснований, и здесь это сделано лишь для того, чтобы продемонстрировать внутренний механизм интерпретации этих директив в Angular.

Анализируя конкретную модель DOM в этих примерах применения директив **ngFor** и **ngIf**, можно заметить, что Angular автоматически вводит в нее HTML-комментарии, описывающие порядок интерпретации разметки и ее преобразования в привязки шаблонов.

Дополнение

Элемент разметки template обязан своим происхождением спецификации веб-компонентов (Web Components). А шаблоны служат для определения

поддерева модели DOM в конечном счете как одного целого, но его элементы не создаются и не активизируются до тех пор, пока из этого шаблона не будет получен экземпляр. Веб-компоненты поддерживаются не во всех браузерах, и поэтому в Angular 2 реализованы полифилы для моделирования надлежащего поведения шаблона.

Подобным образом директива `ngFor` фактически создает шаблон веб-компонента, использующий подчиненную привязку по директиве `ngForOf`, являющейся свойством класса `NgFor`. В каждом экземпляре из массива `articles` будет использоваться шаблон для создания раздела в модели DOM, и в этом разделе переменные шаблона `article` и `index` будут доступны для вставки.

См. также

- Рецепт "Применение декораторов для создания и стилизации простого компонента", описывающий стандартные блоки для реализации компонентов в Angular 2.
- Рецепт "Передача членов родительского компонента порожденному компоненту", поясняющий основы организации нисходящего потока данных между компонентами.
- Рецепт "Привязка к естественным атрибутам элементов разметки", в котором показано, каким образом Angular 2 сопрягается с атрибутами элементов HTML-разметки.
- Рецепт "Присоединение поведения к элементам модели DOM с помощью директив", демонстрирующий, как присоединять поведение к элементам с помощью директив атрибутов.
- Рецепт "Привязка свойств атрибутов", в котором представлен искусный способ углубленного обращения к свойствам элементов разметки в Angular 2.
- Рецепт "Применение перехватчиков жизненного цикла компонентов", в котором приводится пример интеграции с потоком визуализации компонентов в Angular 2.

Обращение к элементам разметки с помощью переменных шаблона

Начиная изучать Angular 2, многие разработчики доходят до чего-то, напоминающего заслуживающую доверия директиву `ng-model` в Angular 1. И это директива `ngModel`. Но в то же время имеется новый способ обращения к элементам разметки в шаблоне с помощью локальных переменных шаблона.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/5094/**.

Подготовка

Допустим, имеется следующее приложение, котором требуется непосредственный доступ к элементу разметки `input`:

```
[app/article.component.ts]

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

@Component({
  selector: 'article',
  template: `
    <input>
    <h1>{{title}}</h1>
  `
})
export class ArticleComponent {}
```

Реализация

В каркасе Angular 2 допускается присваивание с помощью знака **#** в самом шаблоне, чтобы на него можно было впоследствии ссылаться из шаблона, как демонстрируется в следующем фрагменте кода:

```
[app/article.component.ts]

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

@Component({
  selector: 'article',
  template: `
    <input #title>
    <h1>{{title}}</h1>
  `
})
export class ArticleComponent {}
```

Таким образом, обозначение `[object HTMLInputElement]` (или нечто подобное в зависимости от конкретного браузера) вставляется в дескриптор `<h1>`. Это означает, что ссылка `#title` в дескрипторе `<input>` теперь делается непосредственно на объект элемента разметки, а следовательно, доступным должно стать значение в этом элементе разметки.

Но не следует пока что обольщаться! Если попытаться вставить свойство `title.value`, а затем манипулировать полем ввода данных, то никакого

обновления в браузере не произойдет. Дело в том, что в Angular 2 больше не поддерживается двунаправленная привязка данных подобным образом. Впрочем, выход из этого затруднения можно найти в новом для Angular 2 шаблоне привязки данных.

Обновление модели DOM не происходит в Angular до тех пор, пока в этом не возникнет насущная потребность. Она определяется таким поведением в приложении, которое может вызвать изменение во вставляемых данных. В частности, привязанное событие, распространяемое вверх по иерархическому дереву компонентов, может вызвать изменение в данных. Таким образом, событие можно привязать к элементу разметки, и самого лишь наличия этой привязки события должно быть достаточно, чтобы шаблон обновился средствами Angular, как показано ниже.

```
[app/article.component.ts]

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

@Component({
  selector: 'article',
  template: `
    <input #title (keyup)="0">
    <h1>{{title.value}}</h1>
  `
})
export class ArticleComponent {}
```

Здесь событие keyup из элемента разметки ввода текста привязано к выражению, которое, по существу, является пустым. А поскольку событие инициирует обновление модели DOM, то можно успешно извлечь последнее свойство value из объекта title, представляющего элемент разметки ввода текста. Таким образом, входное значение будет успешно привязано к вставляемой символьной строке.

Дополнение

Если вас не особенно устраивает обозначение **#**, можете всегда заменить его префиксом **val-**, чтобы добиться того же самого поведения:

```
[app/article.component.ts]

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

@Component({
  selector: 'article',
  template: `
    <input val-title (keyup)="0">
    <h1>{{title.value}}</h1>
```

```
  `
})
export class ArticleComponent {}
```

Важно также вспомнить, что эти переменные шаблона доступны только в пределах шаблона. Если же требуется передать их обратно контроллеру, его придется указать в качестве аргумента обработчика событий:

```
[app/article.component.ts]

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

@Component({
  selector: 'article',
  template: `
    <input #title (keyup)="setTitle(title.value)">
    <h1>{{myTitle}}</h1>
  `
})
export class ArticleComponent {
  myTitle:string;

  setTitle(val:string):void {
    this.myTitle = val;
  }
}
```

См. также

- Рецепт "Обращение к родительскому компоненту из порожденного компонента", в котором описывается, каким образом компонент может получить прямую ссылку на своего родителя через внедрение зависимостей.
- Рецепт "Настройка взаимного оповещения родительских и порожденных компонентов с помощью декоратора `@ViewChild` и функции `forwardRef()`", в котором даются советы о том, как пользоваться декоратором `@ViewChild` для ссылки на экземпляры объектов порожденных компонентов.
- Рецепт "Настройка взаимного оповещения родительских и порожденных компонентов с помощью декоратора `@ContentChild` и функции `forwardRef()`", в котором даются советы о том, как пользоваться декоратором `ContentChild` и функцией `forwardRef()` для ссылки на экземпляры объектов порожденных компонентов.

Привязка свойств атрибутов

К числу самых главных преимуществ нового стиля привязки в Angular относится возможность более точно намечать то, что привязывается. Прежде атрибут HTML-разметки применялся в качестве директивы или маркера данных, а проще говоря, как согласующий идентификатор. А теперь привязку свойств можно употребить в разметке привязки как для ввода, так и для вывода.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/8565/**.

Подготовка

Допустим, имеется следующее приложение:

```
[app/article.component.ts]

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

@Component({
  selector: 'article',
  template: `
    <input #title (keydown)="setTitle(title.value)">
    <h1>{{myTitle}}</h1>
  `
})
export class ArticleComponent {
  myTitle:string;

  setTitle(val:string):void {
    this.myTitle = val;
  }
}
```

Цель в данном случае состоит в том, чтобы, модифицировав это приложение, придать ему следующее поведение.

- Дескриптор <h1> не будет обновляться значением из поля ввода данных до тех пор, пока пользователь не нажмет клавишу <Enter>.
- Если значение в дескрипторе <h1> не совпадает со значением в поле ввода названия (назовем это состояние "устаревшим"), то текст должен быть выделен красным цветом. А если такое совпадение имеет место, то текст выделяется зеленым цветом.

Реализация

Оба упомянутых выше вида поведения могут быть достигнуты путем привязки свойств атрибутов в Angular 2. Прежде всего, привязку событий можно изменить таким образом, чтобы к обратному вызову приводило нажатие клавиши <Enter>, как выделено ниже полужирным шрифтом.

```
[app/article.component.ts]

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

@Component({
  selector: 'article',
  template: `
    <input #title (keydown.enter)="setTitle(title.value)">
    <h1>{{myTitle}}</h1>
  `
})
export class ArticleComponent {
  myTitle:string;
  setTitle(val:string):void {
    this.myTitle = val;
  }
}
```

Далее можно воспользоваться принятым в Angular стилем привязки, чтобы непосредственно присвоить значение свойству `style`. Чтобы сохранить состояние, в объект контроллера придется добавить логическое значение следующим образом:

```
[app/article.component.ts]

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

@Component({
  selector: 'article',
  template: `
    <input #title (keydown.enter)="setTitle(title.value)">
    <h1 [style.color]="isStale ? 'red' : 'green'">{{myTitle}}</h1>
  `
})
export class ArticleComponent {
  myTitle:string = '';
  isStale:boolean = false;
  setTitle(val:string):void {
    this.myTitle = val;
  }
}
```

Но при ближайшем рассмотрении это все еще не дает возможности получить устаревшее состояние. Чтобы добиться этого, добавьте еще одну привязку к событию keydown:

```
[app/article.component.ts]

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

@Component({
  selector: 'article',
  template: `
    <input #title
           (keyup.enter)="setTitle(title.value)"
           (keyup)="checkStale(title.value)">
    <h1 [style.color]="isStale ? 'red' : 'green'">
      {{myTitle}}
    </h1>
  `
})
export class ArticleComponent {
  myTitle:string = '';
  private isStale:boolean = false;
  setTitle(val:string):void {
    this.myTitle = val;
  }
  checkStale(val:string):void {
    this.isStale = val !== this.myTitle;
  }
}
```

Таким образом, цвет в дескрипторе <h1> позволяет правильно отслеживать, устарели ли данные или нет!

Принцип действия

Если пояснить просто, то в Angular предоставляется немало синтаксических удобств, хотя и на самом элементарном уровне и без излишней сложности. А если исследовать событие keyup, то можно обнаружить в нем отсутствие свойства enter. В каркасе Angular предоставляется немало подобных псевдосвойств, поэтому проверять нажатие клавиши в свойстве keyCode необязательно.

Аналогично в Angular допускаются также привязка и непосредственный доступ к свойствам style. Из этого следует, что стиль, к которому выполняется обращение, будет относиться к основному элементу разметки.

Дополнение

Следует иметь в виду, что здесь были назначены два обработчика для одного и того же события. И не только это, но и изменение порядка привязки разметки способно нарушить требуемое поведение данного приложения.

Если для одного и того же события назначаются два обработчика, то Angular выполнит их в том порядке, в каком они определены в разметке.

См. также

- Рецепт "Привязка к естественным атрибутам элементов разметки", демонстрирующий, каким образом Angular 2 сопрягается с атрибутами элементов HTML-разметки.
- Рецепт "Регистрация обработчиков естественных событий в браузерах", в котором показано, насколько просто присоединить поведение к событиям в браузере.
- Рецепт "Генерирование и обработка специальных событий средствами класса `EventEmitter`", подробно поясняющий, каким образом информация распространяется вверх по иерархии компонентов.
- Рецепт "Присоединение поведения к элементам модели DOM с помощью директив", демонстрирующий, как присоединять поведение к элементам с помощью директив атрибутов.

Применение перехватчиков жизненного цикла компонентов

Процессу визуализации компонентов в Angular присуще немало особенностей, и в разные моменты времени становятся доступными разные типы данных и ссылки. Чтобы учесть все это, в компонентах Angular 2 допускается устанавливать обратные вызовы, которые будут выполняться в разных точках жизненного цикла компонента.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **`http://ngcookbook.herokuapp.com/2048/`**.

Подготовка

Допустим, что разработка была начата с приведенного ниже приложения, в котором просто вводятся и удаляются статьи, получаемые из одних входных данных. Цель состоит в том, чтобы воспользоваться перехватчиками

жизненного цикла компонентов и таким образом отслеживать процесс операций ввода и удаления.

```
[app/article-list.component.ts]

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

@Component({
  selector: 'article-list',
  template: `
    <input (keyup.enter)="add($event)">
    <article *ngFor="let title of titles; let i = index"
             [articleTitle]="title">
         <button (click)="remove(i)">X</button>
    </article>
  `
})
export class ArticleListComponent {
  titles:Array<string> = [];
  add(e:Event):void {
    this.titles.push(e.target.value);
    e.target.value = '';
  }
  remove(index:number) {
    this.titles.splice(index, 1);
  }
}
[app/article.component.ts]

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

@Component({
  selector: 'article',
  template: `
    <h1>
      <ng-content></ng-content>{{articleTitle}}
    </h1>
  `
})
export class ArticleComponent {
  @Input() articleTitle:string;
}
```

Реализация

В каркасе Angular допускается импортировать интерфейсы перехвата из базового модуля. Эти интерфейсы проявляются в виде методов класса, вызываемых в подходящий момент, как показано ниже. Таким образом, сообщения, фиксируемые в журнале, должны появляться всякий раз, когда вводится или удаляется новый компонент `ArticleComponent`.

```
[app/article.component.ts]

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

@Component({
  selector: 'article',
  template: `
    <h1>
      <ng-content></ng-content>{{articleTitle}}
    </h1>
  `
})
export class ArticleComponent implements OnInit, OnDestroy {
  @Input() articleTitle:string;
  ngOnInit() {
    console.log('created', this.articleTitle);
  }
  ngOnDestroy() {
    console.log('destroyed', this.articleTitle);
  }
}
```

Принцип действия

Различные перехватчики имеют разное семантическое назначение, но они будут выполняться во вполне определенном порядке. Выполнение каждого перехватчика гарантирует, что определенное поведение компонента только что состоялось.

Ниже перечислены перехватчики, доступные в порядке их выполнения.

- `ngOnChanges()`
- `ngOnInit()`
- `ngDoCheck()`
- `ngAfterContentInit()`
- `ngAfterContentChecked()`
- `ngAfterViewInit()`
- `ngAfterContentChecked()`
- `ngOnDestroy()`

Перечень этих перехватчиков может быть расширен в сторонних библиотеках, где могут быть введены и собственные перехватчики.

Дополнение

Применять перехватчики совсем не обязательно, и они будут вызываться в Angular лишь в том случае, если они определены. Необязательно употреблять и объявление интерфейса `implements`, хотя оно извещает компилятор TypeScript о том, что следует ожидать соответствующий метод, и это, очевидно, является нормой надлежащей практики.

См. также

- Рецепт "Регистрация обработчиков естественных событий в браузерах", демонстрирующий, насколько просто присоединить поведение к событиям в браузере.
- Рецепт "Настройка взаимного оповещения родительских и порожденных компонентов с помощью декоратора `@ViewChild` и функции `forwardRef()`", в котором даются советы о том, как пользоваться декоратором `@ViewChild` для ссылки на экземпляры объектов порожденных компонентов.
- Рецепт "Настройка взаимного оповещения родительских и порожденных компонентов с помощью декоратора `@ContentChild` и функции `forwardRef()`", в котором даются советы о том, как пользоваться декоратором `@ContentChild` и функцией `forwardRef()` для ссылки на экземпляры объектов порожденных компонентов.

Обращение к родительскому компоненту из порожденного компонента

В процессе написания приложения может возникнуть ситуация, когда было бы удобно обратиться к родительскому компоненту из порожденного компонента, например, для проверки компонентных объектов или вызова открытого метода. Сделать это в Angular 2 совсем не трудно.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **`http://ngcookbook.herokuapp.com/4907/`**.

Подготовка

Допустим, что разработка начата с приведенного ниже компонента `ArticleComponent`. Цель состоит в том, чтобы реализовать компонент обратной связи таким образом, чтобы он отображал количество переданных ему симпатий (likes), а родительский компонент вел подсчет симпатий и передавал его значение.

```
[app/article.component.ts]

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

@Component({
  selector: 'article',
  template: `
    <feedback [val]="likes"></feedback>
  `
})
export class ArticleComponent {
  likes:number = 0;
  incrementLikes():void {
    this.likes++;
  }
}
```

Реализация

Начните с реализации основной структуры порожденного компонента следующим образом:

```
[app/feedback.component.ts]

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

@Component({
  selector: 'feedback',
  template: `
    <h1>Number of likes: {{ val }}</h1>
    <button (click)="likeArticle()">Like this article!</button>
  `
})
export class FeedbackComponent {
  @Input() val:number;
  likeArticle():void {}
}
```

Во всем, что было сделано до сих пор, нет ничего особенного. Если щелкнуть на экранной кнопке, то будет вызван пустой метод, в котором требуется вызвать метод из родительского компонента. Но в настоящий момент недостает ссылки, чтобы сделать это. А если указать компонент в конструкторе порожденного компонента, то он станет доступным, как показано ниже.

```
[app/feedback.component.ts]

import {Component, Input} from '@angular/core';
import {ArticleComponent} from './article.component';

@Component({
  selector: 'feedback',
```

```
  template: `
    <h1>Number of likes: {{ val }}</h1>
    <button (click)="like()">Like this article!</button>
  `
})
export class FeedbackComponent {
  @Input() val:number;
  private articleComponent:ArticleComponent;
  constructor(articleComponent:ArticleComponent) {
    this.articleComponent = articleComponent;
  }
  like():void {
    this.articleComponent.incrementLikes();
  }
}
```

Теперь, имея в своем распоряжении ссылку на родительский компонент, можно вызвать его открытый метод, а именно `incrementLikes()`. В данный момент оба компонента должны взаимодействовать правильно.

Принцип действия

Проще говоря, в Angular 2 распознается, что вы внедряете компонент, типизированный таким же образом, как и родительский, и в конечном счете этот родительский компонент предоставляется в ваше распоряжение. Таким образом, родительский компонент полностью наследуется, и вы вольны взаимодействовать с ним, как с экземпляром любого другого компонента.

Ссылку на компонент требуется хранить в конструкторе. В отличие от внедрения службы, порожденный компонент не сделает доступным экземпляр компонента типа **`ArticleComponent`** по ссылке **`this.articleComponent`**. Следовательно, это необходимо сделать вручную.

Дополнение

Проницательный разработчик заметит, что в данном случае образуется очень жесткая зависимость между порожденным и родительским компонентами. И это действительно так, хотя и совсем не обязательно. Зачастую оказывается удобным предоставить компонентам возможность более просто взаимодействовать друг с другом за счет их модульности. И, как правило, это будет субъективным решением с вашей стороны.

См. также

- Рецепт "Передача членов родительского компонента порожденному компоненту", поясняющий основы организации нисходящего потока данных между компонентами.

- Рецепт "Регистрация обработчиков естественных событий в браузерах", в котором показано, насколько просто присоединить поведение к событиям в браузере.
- Рецепт "Генерирование и обработка специальных событий средствами класса `EventEmitter`", подробно поясняющий, каким образом информация распространяется вверх по иерархии компонентов.
- Рецепт "Настройка взаимного оповещения родительских и порожденных компонентов с помощью декоратора `@ViewChild` и функции `forwardRef()`", в котором даются советы о том, как пользоваться декоратором `@ViewChild` для ссылки на экземпляры объектов порожденных компонентов.
- Рецепт "Настройка взаимного оповещения родительских и порожденных компонентов с помощью декоратора `@ContentChild` и функции `forwardRef()`", в котором даются советы о том, как пользоваться декоратором `@ContentChild` и функцией `forwardRef()` для ссылки на экземпляры объектов порожденных компонентов.

Настройка взаимного оповещения родительских и порожденных компонентов с помощью декоратора `@ViewChild` и функции `forwardRef()`

В зависимости от степени разделения ответственности в приложении имеет смысл организовать в порожденном компоненте обращение к родительскому компоненту и одновременно — к порожденному компоненту из родительского. И этого можно добиться с помощью декораторов `@ViewChild` и `@ContentChild`. Оба эти декоратора обсуждаются в данном и последующем рецептах.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **`http://ngcookbook.herokuapp.com/1315/`**.

Подготовка

Начните с настройки приложения согласно рецепту "Обращение к элементам разметки с помощью переменных шаблона". Цель состоит в том, чтобы добавить возможность активировать и деактивировать экранную кнопку выбора симпатий из родительского компонента.

Реализация

В первоначальной настройке предоставляется доступ только из порожденного к родительскому компоненту, но это лишь половина того, чего требуется добиться. Требуется также предоставить доступ к порожденному компоненту. Получить ссылку на компонент `FeedbackComponent`, наблюдаемый в представлении шаблона из компонента `ArticleComponent`, можно двумя способами. И первый демонстрируемый здесь способ состоит в применении декоратора `@ViewChild`.

Настройка ссылки в декораторе `@ViewChild`

Используя декоратор `@ViewChild`, можно извлечь ссылку на компонент из представления. Проще говоря, применение декоратора `@ViewChild` в данном примере дает возможность ссылаться на экземпляр компонента `FeedbackComponent` из кода в компоненте `ArticleComponent`.

Прежде всего настройте компонент `ArticleComponent` таким образом, чтобы извлекать в нем ссылку на другой компонент:

```
[app/article.component.ts]

import {Component, ViewChild} from '@angular/core';
import {FeedbackComponent} from './feedback.component';

@Component({
  selector: 'article',
  template: `
    <input type="checkbox"
           (click)="changeLikesEnabled($event)">
    <feedback [val]="likes"></feedback>
  `
})
export class ArticleComponent {
  @ViewChild(FeedbackComponent)
       feedbackComponent:FeedbackComponent;
  likes:number = 0;
  incrementLikes():void {
    this.likes++;
  }
  changeLikesEnabled(e:Event):void {
     this.feedbackComponent.setLikeEnabled(e.target.checked);
  }
}
```

Главная особенность этого нового кода заключается в том, что в декораторе `@ViewChild` неявно подразумевается, что он должен наметить представление данного компонента, найти визуализируемый здесь экземпляр типа

FeedbackComponent и присвоить его члену feedbackComponent экземпляра типа ArticleComponent.

Исправление цикла зависимостей с помощью функции forwardRef()

В данный момент вы должны заметить, что ваше приложение сгенерирует новые ошибки, главным образом, из-за неспособности разрешить параметры для компонента FeedbackComponent. И это происходит потому, что вы установили циклическую зависимость компонента FeedbackComponent от компонента ArticleComponent, а последнего — от первого. Правда, данное затруднение существует и в области внедрения зависимостей в Angular, и поэтому на самом деле требуется не модуль, а только представляющий его маркер. Именно по этой причине в Angular 2 предоставляется функция forwardRef(), допускающая пользоваться в определении класса зависимостью от модуля, прежде чем она будет определена. Воспользуйтесь этой функцией следующим образом:

```
[app/feedback.component.ts]

import {Component, Input, Inject, forwardRef}
        from '@angular/core';
import {ArticleComponent} from './article.component';

@Component({
  selector: 'feedback',
  template: `
    <h1>Number of likes: {{ val }}</h1>
     <button (click)="likeArticle()">
        Like this article!
     </button>
  `
})
export class FeedbackComponent {
  @Input() val:number;
  private articleComponent:ArticleComponent;
  constructor(@Inject(forwardRef(() => ArticleComponent))
              articleComponent:ArticleComponent) {
    this.articleComponent = articleComponent;
  }
  likeArticle():void {
    this.articleComponent.incrementLikes();
  }
}
```

Добавление желаемого поведения

Разрешив циклическую зависимость, добавьте метод setLikeEnabled(), вызываемый в родительском компоненте, как показано ниже. Таким образом,

установкой или сбросом флажка должна активироваться или деактивироваться экранная кнопка выбора симпатий.

```
[app/feedback.component.ts]

import {Component, Input, Inject, forwardRef}
        from '@angular/core';
import {ArticleComponent} from './article.component';

@Component({
  selector: 'feedback',
  template: `
    <h1>Number of likes: {{ val }}</h1>
    <button (click)="likeArticle()"
            [disabled]="!likeEnabled">
      Like this article!
    </button>
  `
})
export class FeedbackComponent {
  @Input() val:number;
  private likeEnabled:boolean = false;
  private articleComponent:ArticleComponent;
  constructor(@Inject(forwardRef(() => ArticleComponent))
              articleComponent:ArticleComponent) {
    this.articleComponent = articleComponent;
  }
  likeArticle():void {
    this.articleComponent.incrementLikes();
  }
  setLikeEnabled(newEnabledStatus:boolean):void {
    this.likeEnabled = newEnabledStatus;
  }
}
```

Принцип действия

Декоратор `@ViewChild` предписывает Angular найти первый экземпляр типа `FeedbackComponent`, имеющийся в представлении компонента `Article Component`, и присвоить его декорированному члену класса. Ссылка будет обновляться вместе с любыми представлениями, а декорированный член — ссылаться на экземпляр порожденного компонента. С ним можно взаимодействовать так же, как и с любым обычным экземпляром объекта.

Очень важно не забывать о двойственности экземпляра компонента и его представления в шаблоне. Например, компонент **FeedbackComponent** представлен дескриптором обратной связи (**pre-render**) и дескриптором заголовка и экранной кнопки (**post-render**), но ни одна из этих форм не

образует конкретный компонент. Экземпляр типа **FeedbackComponent** является объектом языка JavaScript, существующим в оперативной памяти. Именно к этому объекту и требуется доступ. Если же потребуется только ссылка на элементы шаблона, ее можно получить, например, с помощью переменной шаблона.

Дополнение

В каркасе Angular выполняется иерархическая визуализация, и поэтому декоратор `@ViewChild` не будет готов до тех пор, пока не инициализируется представление, а по существу, после перехватчика жизненного цикла компонента `AfterViewInit`. Это положение можно продемонстрировать следующим образом:

```
[app/article.component.ts]

import {Component, ViewChild, ngAfterViewInit}
        from '@angular/core';
import {FeedbackComponent} from './feedback.component';

@Component({
  selector: 'article',
  template: `
    <input type="checkbox"
           (click)="changeLikesEnabled($event)">
    <feedback [val]="likes"></feedback>
  `
})
export class ArticleComponent implements AfterViewInit {
  @ViewChild(FeedbackComponent)
       feedbackComponent:FeedbackComponent;
  likes:number = 0;
  constructor() {
    console.log(this.feedbackComponent);
  }
  ngAfterViewInit() {
    console.log(this.feedbackComponent);
  }
  incrementLikes():void {
    this.likes++;
  }
  changeLikesEnabled(e:Event):void {
    this.feedbackComponent.setLikeEnabled(e.target.checked);
  }
}
```

Первоначально вывод в журнал в виде представления окажется неопределенным в конструкторе, а следовательно, компонент `FeedbackComponent`

пока еще не существует. Как только произойдет перехват жизненного цикла компонента `AfterViewInit`, появится возможность увидеть вывод в журнал из компонента `FeedbackComponent` на консоль.

Декоратор `@ViewChildren`

Если требуется получить ссылку на несколько компонентов, идентичную ссылку можно приобрести с помощью декоратора `@ViewChildren`. Этот декоратор предоставит список типа `QueryList` всех совпадающих компонентов в представлении.

Список типа **`QueryList`** может быть использован как массив с методом **`toArray()`**. Он обеспечивает доступ к свойству **`changes`**, обусловливающему распространение события всякий раз, когда изменяется элемент в списке типа **`QueryList`**.

См. также

- Рецепт "Применение перехватчиков жизненного цикла компонентов", в котором приводится пример интеграции с потоком визуализации компонентов в Angular 2.
- Рецепт "Обращение к элементам разметки с помощью переменных шаблона", демонстрирующий новую конструкцию переменных шаблона в Angular 2.
- Рецепт "Настройка взаимного оповещения родительских и порожденных компонентов с помощью декоратора `@ContentChild` и функции `forwardRef()`", в котором даются советы о том, как пользоваться декоратором `@ContentChild` и функцией `forwardRef()` для ссылки на экземпляры объектов порожденных компонентов.

Настройка взаимного оповещения родительских и порожденных компонентов с помощью декоратора `@ContentChild` и функции `forwardRef()`

Под стать декоратору `@ViewChild` в Angular оказывается и декоратор `@ContentChild`. Он выполняет аналогичную обязанность, получая ссылку на целевой порожденный компонент и делая его доступным в качестве члена экземпляра родительского компонента. Отличие заключается в том, что декоратор `@ContentChild` извлекает разметку, существующую в дескрипторах селектора родительского компонента, тогда как декоратор `@ViewChild` извлекает разметку, существующую в преставлении родительского компонента.

Данное отличие лучше всего продемонстрировать, сравнив поведение обоих декораторов. С этой целью здесь будет переделан пример из рецепта "Настройка взаимного оповещения родительских и порожденных компонентов с помощью декоратора `@ViewChild` и функции `forwardRef()`", чтобы воспользоваться в нем декоратором `@ContentChild`.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/7386/**.

Подготовка

Начните с исходного кода из примера, представленного в рецепте "Настройка взаимного оповещения родительских и порожденных компонентов с помощью декоратора `@ViewChild` и функции `forwardRef()`".

Реализация

Прежде чем приступить к переделке упомянутого выше примера, необходимо вложить дескрипторы компонента `ArticleComponent` в другой корневой компонент, поскольку декоратор `@ContentChild` не пригоден для компонента приложения, запускаемого на корневом уровне. С этой целью создайте заключаемый в оболочку компонент `RootComponent`, как показано ниже.

```
[app/root.component.ts]

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

@Component({
  selector: 'root',
  template: `
    <article></article>
  `
})
export class RootComponent {}
```

Переход к декоратору `@ContentChild`

Декоратор `@ContentChild` внедряется в компоненты, таким же образом, как и декоратор `@ViewChild`. Итак, перейдите к декоратору `@ContentChild` в самом компоненте `ArticleComponent`, заменив дескриптор `<feedback>` на дескриптор `<ng-content>` следующим образом:

```
[app/article.component.ts]

import {Component, ContentChild} from '@angular/core';
import {FeedbackComponent} from './feedback.component';
```

```
@Component({
  selector: 'article',
  template: `
    <input type="checkbox"
           (click)="changeLikesEnabled($event)">
    <ng-content></ng-content>
  `
})
export class ArticleComponent {
  @ContentChild(FeedbackComponent)
      feedbackComponent:FeedbackComponent;
  likes:number = 0;
  incrementLikes():void {
    this.likes++;
  }
  changeLikesEnabled(e:Event):void {
    this.feedbackComponent.setLikeEnabled(e.target.checked);
  }
}
```

Таким образом, порожденный компонент, безусловно, может быть найден лишь в том случае, если в дескрипторе <article></article> находится его содержимое, как выделено ниже полужирным шрифтом.

```
[app/root.component.ts]

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

@Component({
  selector: 'root',
  template: `
    <article>
      <feedback></feedback>
    </article>
  `
})
export class RootComponent {}
```

Обратите внимание на то, что значение счетчика симпатий, передаваемое порожденному компоненту в качестве входных данных, было удалено. Говоря коротко, данное соглашение здесь больше не действует, поскольку его привязка приведет к извлечению счетчика симпатий из компонента **RootComponent**, где подобная информация отсутствует.

Исправление привязки данных

В компоненте FeedbackComponent потребуется извлечь значение счетчика симпатий непосредственно:

```
[app/feedback.component.ts]

import {Component, Inject, forwardRef} from '@angular/core';
import {ArticleComponent} from './article.component';

@Component({
  selector: 'feedback',
  template: `
    <h1>Number of likes: {{ val }}</h1>
    <button (click)="likeArticle()"
            [disabled]="!likeEnabled">
      Like this article!
    </button>
  `
})
export class FeedbackComponent {
  private val:number;
  private likeEnabled:boolean = false;
  private articleComponent:ArticleComponent;
  constructor(@Inject(forwardRef(() => ArticleComponent))
              articleComponent:ArticleComponent) {
    this.articleComponent = articleComponent;
    this.updateLikes();
  }
  updateLikes() {
    this.val = this.articleComponent.likes;
  }
  likeArticle():void {
    this.articleComponent.incrementLikes();
    this.updateLikes();
  }
  setLikeEnabled(newEnabledStatus:boolean):void {
    this.likeEnabled = newEnabledStatus;
  }
}
```

Вот и все! Теперь приложение должно вести себя идентично настройке, сделанной ранее в разделе "Подготовка" данного рецепта.

Принцип действия

Декоратор `@ContentChild` делает почти то же самое, что и декоратор `@ViewChild`. Он лишь выполняет поиск в другом месте. В частности, декоратор `@ContentChild` предписывает Angular найти первый экземпляр типа `FeedbackComponent`, имеющийся в дескрипторах компонента `ArticleComponent`. Сначала делается ссылка на то, что вставлено в дескриптор `<ng-content>`. Затем найденный экземпляр компонента присваивается декорированному члену класса. Полученная ссылка обновляется вместе с любыми представлениями. В итоге декорированный член будет ссылаться на

экземпляр порожденного компонента, допуская взаимодействие с ним, как и с любым обычным экземпляром объекта.

Дополнение

В каркасе Angular выполняется иерархическая визуализация, и поэтому декоратор `@ContentChild` не будет готов до тех пор, пока не инициализируется представление, а по существу, после работы перехватчика жизненного цикла компонента `AfterContentInit`. Это положение можно продемонстрировать следующим образом:

```
[app/article.component.ts]

import {Component, ContentChild, ngAfterContentInit}
        from '@angular/core';
import {FeedbackComponent} from './feedback.component';

@Component({
  selector: 'article',
  template: `
    <input type="checkbox"
           (click)="changeLikesEnabled($event)">
    <ng-content></ng-content>
  `
})
export class ArticleComponent implements AfterContentInit {
  @ContentChild(FeedbackComponent)
          feedbackComponent:FeedbackComponent;
  likes:number = 0;
  constructor() {
    console.log(this.feedbackComponent);
  }
  ngAfterContentInit() {
    console.log(this.feedbackComponent);
  }
  incrementLikes():void {
    this.likes++;
  }
  changeLikesEnabled(e:Event):void {
    this.feedbackComponent.setLikeEnabled(e.target.checked);
  }
}
```

Первоначально вывод в журнал содержимого окажется неопределенным в конструкторе, а следовательно, компонент `FeedbackComponent` пока еще не существует. Как только произойдет перехват жизненного цикла компонента `AfterContentInit`, появится возможность увидеть вывод в журнал компонента `FeedbackComponent` на консоль.

Декоратор `@ContentChildren`

Если требуется получить ссылку на несколько компонентов, идентичную ссылку можно приобрести с помощью декоратора `@ContentChildren`. Этот декоратор предоставит список типа `QueryList` всех совпадающих компонентов в дескрипторах компонента.

Список типа **`QueryList`** может быть использован как массив с методом **`toArray()`**. Он обеспечивает доступ к свойству **`changes`**, обусловливающему распространение события всякий раз, когда изменяется элемент в списке типа **`QueryList`**.

См. также

- Рецепт "Применение перехватчиков жизненного цикла компонентов", в котором приводится пример интеграции с потоком визуализации компонентов в Angular 2.
- Рецепт "Обращение к элементам разметки с помощью переменных шаблона", демонстрирующий новую конструкцию переменных шаблона в Angular 2.
- Рецепт "Настройка взаимного оповещения родительских и порожденных компонентов с помощью декоратора `@ViewChild` и функции `forwardRef()`", в котором даются советы о том, как пользоваться декоратором `@ContentChild` и функцией `forwardRef()` для ссылки на экземпляры объектов порожденных компонентов.

Глава 3

Создание реактивных и управляемых по шаблонам форм

В этой главе даются следующие рецепты.

- Реализация простой двунаправленной привязки данных с помощью директивы `ngModel`
- Реализация элементарной проверки достоверности данных в поле средствами класса `FormControl`
- Объединение объектов типа `FormControl` средствами класса `FormGroup`
- Объединение объектов типа `FormControl` средствами класса `FormArray`
- Реализация элементарных форм с помощью директивы `ngForm`
- Реализация элементарных форм средствами класса `FormBuilder` и директивы `formControlName`
- Создание и применение специального средства проверки достоверности
- Создание и применение асинхронного средства проверки достоверности с помощью обязательств

Введение

Формы являются важными элементарными конструкциями практически для каждого веб-приложения, и они были переосмыслены и усовершенствованы в Angular 2. Формы оказались очень полезными в Angular 1, но они были полностью зависимы от соглашений, принятых в директиве `ngModel`. Вновь принятые в Angular 2 соглашения устраняют подобную зависимость и предлагают свежий, более ясный и удобный подход к формам и управлению информацией.

По сути, важно уяснить, где и почему формы оказываются удобными. В приложении имеется немало мест, где многообразные входные данные требуют ассоциации, и в этом контексте формы, безусловно, оказываются удобными. В каркасе Angular 2 формы лучше всего подходят для проверки достоверности упомянутых выше входных данных, особенно в том случае, если требуется проверить достоверность данных в нескольких полях или сделать это перекрестным способом. Кроме того, в Angular поддерживается состояние различных элементов форм, что дает пользователю возможность анализировать "предысторию" ввода данных в каждом поле.

Не следует также забывать, что поведение формы в Angular 2, как, впрочем, и привязка событий и данных, тесно связаны с уже отработанным в браузере поведением формы. Браузеры вполне приспособлены к передаче данных на обработку, восстанавливая их после загрузки страницы, простой проверки достоверности и прочих видов поведения, которые используются почти во всех формах. Виды поведения в Angular 2 не переопределяются, а, напротив, сочетаются с видами поведения в браузере, чтобы устанавливать связь с другими видами поведения и данными, являющимися частью каркаса или приложения.

В этой главе обращает на себя внимание двойственность применения модулей **`FormsModule`** и **`ReactiveFormsModule`**. Они ведут себя совсем иначе и почти всегда применяются раздельно, когда дело доходит для создания формы.

Реализация простой двунаправленной привязки данных с помощью директивы `ngModel`

В каркасе Angular 2 по-прежнему поддерживается двунаправленная привязка данных, но действует она несколько иначе, чем к этому привыкли пользователи Angular. Вначале данного рецепта демонстрируется очень простой пример, который разбирается на части для описания того, что на самом деле происходит во время привязки.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **`http://ngcookbook.herokuapp.com/0771/`**.

Реализация

Для двунаправленной привязки служит директива `ngModel`, входящая в состав модуля `FormsModule`. Итак, добавьте эту директиву в модуль своего приложения следующим образом:

```
[app/app.module.ts]

import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {FormsModule} from '@angular/forms';
import {ArticleEditorComponent} from './article-editor.component';

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

Далее реализуйте свой компонент, привязав два экземпляра элементов ввода данных к одному и тому же члену компонента с помощью директивы ngModel:

```
[app/article-editor.component.ts]

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

@Component({
  selector: 'article-editor',
  template: `
    <input [(ngModel)]="title">
    <input [(ngModel)]="title">
    <h2>{{title}}</h2>
  `
})
export class ArticleEditorComponent {
  title:string;
}
```

Вот, собственно, и все, что требуется! Таким образом, любые изменения в одних элементах ввода данных будут сразу же отражаться в других элементах, а также в заголовке, описываемом дескриптором <h2>.

Принцип действия

На самом деле вы осуществляете привязку события и свойства, которое директива ассоциирует с конкретным элементом ввода данных. Когда в компоненте изменяется член title, элемент ввода данных привязывается

к значению этого члена и соответственно обновляет свое значение. Когда же изменяется значение в элементе ввода данных, он отправляет событие, а директива ngModel привязывает к нему это событие, извлекая из него значение, прежде чем распространить его к члену title данного компонента.

Синтаксис *вложенных скобок* **[()]**, похожий на бананы в ящике, обозначает привязку как к свойству ввода данных (квадратные скобки), так и к событиям ввода данных (круглые скобки).

В действительности это сокращенная форма выделенной ниже полужирным шрифтом формы привязки данных. Можете сами убедиться, что она действует таким же образом, как и описанная выше.

```
[app/article-editor.component.ts]

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

@Component({
  selector: 'article-editor',
  template: `
    <input [ngModel]="title" (ngModelChange)="title=$event">
    <input [ngModel]="title" (ngModelChange)="title=$event">
    <h2>{{title}}</h2>
  `
})
export class ArticleEditorComponent {
  title:string;
}
```

Дополнение

Такое синтаксическое удобство может показаться вам чрезмерным. Вы осуществляете привязку к событию по директиве ngModel, но каким-то образом это оказывается равнозначным привязке к входному значению. Аналогично вы осуществляете привязку к событиям по директиве ngModelChange, и все они отправляют параметр $event, который оказывается всего лишь символьной строкой. На самом деле все верно. Директива ngModel различает отдельные части [ngModel] и (ngModelChange), правильно объединяя их, чтобы ассоциировать с требующимися привязками.

По существу, в самих привязках делается следующее:

```
[app/article-editor.component.ts]

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

@Component({
  selector: 'article-editor',
  template: `
```

```
    <input [value]="title" (input)="title=$event.target.value">
    <input [value]="title" (input)="title=$event.target.value">
    <h2>{{title}}</h2>
  `
})
export class ArticleEditorComponent {
  // инициализировать свойство title, иначе будет получено
  // сообщение, что оно не определено (undefined)
  title:string = '';
}
```

См. также

- Рецепт "Реализация элементарной проверки достоверности данных в поле средствами класса FormControl", в котором подробно описывается элементарный блок привязки в форме Angular.
- Рецепт "Объединение объектов типа FormControl средствами класса FormGroup", демонстрирующий, как объединять объекты типа FormControl.
- Рецепт "Объединение объектов типа FormControl средствами класса FormArray" с пояснениями, как обращаться с итерируемыми элементами формы.
- Рецепт "Реализация элементарных форм с помощью директивы ngForm", описывающий декларативный способ создания формы в Angular.
- Рецепт "Реализация элементарных форм средствами класса FormBuilder и директивы formControlName", в котором поясняется, как пользоваться службой FormBuilder для быстрого составления вложенных форм.
- Рецепт "Создание и применение специального средства проверки достоверности", описывающий порядок создания специальной директивы, действующей подобно проверке достоверности.
- Рецепт "Создание и применение асинхронного средства проверки достоверности с помощью обязательств", в котором поясняется, как организовать в Angular отложенную оценку состояния формы.

Реализация элементарной проверки достоверности данных в поле средствами класса FormControl

Самым простым поведением формы, которое можно только представить, является проверка достоверности данных в одном поле их ввода. Чаще всего для этого используют дескрипторы <form> и пишут стереотипный код, но

проверку достоверности единственных вводимых данных лучше всего свести к самому минимуму того, что для этого на самом деле требуется.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/4076/**.

Подготовка

Допустим, имеется приведенный ниже первоначальный вариант приложения. Цель состоит в том, чтобы изменить код таким образом, чтобы после щелчка на экранной кнопке сохранения производилась проверка достоверности введенных данных, а свойство `title` компонента обновлялось лишь в том случае, если эти данные достоверны.

```
[app/article-editor.component.ts]

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

@Component({
  selector: 'article-editor',
  template: `
    <p>Article title (required):</p>
    <input required>
    <button>Save</button>
    <h1>{{title}}</h1>
  `
})
export class ArticleEditorComponent {
  title:string;
}
```

Реализация

Наиболее элементарным компонентом форм в Angular служит объект типа `FormControl`. Чтобы получить доступ к состоянию поля, необходимо сначала получить экземпляр этого объекта в компоненте и связать его с полем, используя директиву `formControl`. Класс `FormControl` входит в состав модуля `ReactiveFormsModule`. Поэтому добавьте в приложение этот модуль, импортировав его, как выделено ниже полужирным шрифтом.

```
[app/app.module.ts]

import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {ReactiveFormsModule} from '@angular/forms';
import {ArticleEditorComponent} from './article-editor.component';

@NgModule({
  imports: [
```

```
    BrowserModule,
    ReactiveFormsModule
  ],
  declarations: [
    ArticleEditorComponent
  ],
  bootstrap: [
    ArticleEditorComponent
  ]
})
export class AppModule {}
```

Таким образом, объект типа FormControl можно применять в компоненте ArticleEditorComponent. Поэтому получите объект типа FormControl в данном компоненте и привяжите его к элементу ввода данных с помощью директивы formControl, как показано ниже.

[app/article-editor.component.ts]

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

@Component({
  selector: 'article-editor',
  template: `
    <p>Article title (required):</p>
    <input [formControl]="titleControl" required>
    <button>Save</button>
    <h1>{{title}}</h1>
  `
})
export class ArticleEditorComponent {
  title:string;
  titleControl:FormControl = new FormControl();
}
```

Создав объект типа FormControl и связав его с полем ввода данных, можете воспользоваться его интерфейсом API для проверки достоверности, чтобы проверить состояние этого поля. И теперь осталось применить его в обработчике событий от щелчков на экранной кнопке, предназначенной для передачи данных на обработку, как показано ниже. В итоге состояние поля ввода данных и значение в нем может быть проверено на достоверность с помощью одного и того же объекта в обработчике событий от щелчков на экранной кнопке, предназначенной для передачи данных на обработку.

[app/article-editor.component.ts]

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

```
@Component({
  selector: 'article-editor',
  template: `
    <p>Article title (required):</p>
    <input [formControl]="titleControl" required>
    <button (click)="submitTitle()">Save</button>
    <h1>{{title}}</h1>
  `
})
export class ArticleEditorComponent {
  title:string;
  titleControl:FormControl = new FormControl();
  submitTitle():void {
    if(this.titleControl.valid) {
      this.title = this.titleControl.value;
    } else {
      alert("Title required");
    }
  }
}
```

Принцип действия

Директива `formControl` служит только для привязки объекта типа `FormControl` к элементу модели DOM. Объект типа `FormControl`, экземпляр которого получается в конструкторе компонента, может использовать атрибуты проверки достоверности в дескрипторе HTML-разметки, как реализовано в данном примере, принять средства проверки достоверности в Angular при его инициализации или же сделать и то и другое.

Крайне важно отметить, что получить экземпляр объекта типа **`FormControl`** явно недостаточно, чтобы сразу же приступить к проверке достоверности введенных данных.

В отсутствие инициализированного значения в пустом поле ввода данных изначально установится пустое значение `null`, которое, естественно, окажется недостоверным при наличии атрибута `required`. Но если бы в данном примере проверить, станет ли объект типа `FormControl` достоверным, как только будет получен экземпляр в конструкторе его класса, то этот объект добросовестно известит, что его состояние достоверно, поскольку он пока еще не привязан к элементу модели DOM. Следовательно, ни одна из проверок достоверности не нарушена. А поскольку привязки элемента ввода данных с помощью директивы `formControl` не произойдет до тех пор, пока шаблон компонента не станет частью конкретной модели DOM, то проверить состояние этого элемента не удастся до завершения привязки или в перехватчике

ngAfterContentChecked() жизненного цикла компонента. Первое из двух упомянутых выше условий относится к рассматриваемому здесь примеру. В частности, как только директива formControl завершит привязку, объект типа FormControl станет оболочкой для введенных данных, давая возможность пользоваться его свойствами valid и value.

Дополнение

В данном рецепте применяется модуль ReactiveFormsModule, который проще понять, поскольку вся установка делается вручную. Если же воспользоваться модулем FormsModule, то можно обнаружить, что большая часть установки по данному рецепту, в том числе получение экземпляров и привязка объектов типа FormControl, может быть сделана автоматически. Действие этого модуля так или иначе связано с наличием дескриптора <form>, который фактически служит контейнером для объектов типа FormControl. А данный рецепт служит для демонстрации одной из самых простых разновидностей поведения форм в Angular.

Средства проверки достоверности и двойственность атрибутов

Как упоминалось ранее в данном рецепте, определения проверки достоверности могут быть сделаны в двух местах. Здесь использован стандартизированный атрибут дескриптора HTML-разметки, распознаваемый и автоматически внедряемый средствами Angular в спецификацию проверки достоверности в классе FormControl. Но для решения той же самой задачи можно было бы с тем же успехом воспользоваться и модулем Validators. Для этого можно было бы сначала импортировать доступные в Angular по умолчанию средства проверки достоверности из модуля Validators, а затем инициализировать объект типа FormControl со средством проверки достоверности required, как показано ниже.

```
[app/article-editor.component.ts]

import {Component} from '@angular/core';
import {FormControl, Validators} from '@angular/forms';

@Component({
  selector: 'article-editor',
  template: `
    <p>Article title (required):</p>
    <input [formControl]="titleControl">
    <button (click)="submitTitle()">Save</button>
    <h1>{{title}}</h1>
  `
})
export class ArticleEditorComponent {
  title:string;
```

```
  // В качестве первого аргумента служит входное значение,
  // устанавливаемое по умолчанию
  titleControl:FormControl =
    new FormControl(null, Validators.required);
  submitTitle():void {
    if(this.titleControl.valid) {
      this.title = this.titleControl.value;
    } else {
      alert("Title required");
    }
  }
}
```

Элементы управления без дескрипторов

Как и следовало ожидать, привязывать объект типа `FormControl` к элементу модели DOM совсем не обязательно. Этот объект является элементарной частицей логики формы, действующей подобно атомарному хранилищу информации о внутреннем состоянии, независимо от того, является ли ее источником дескриптор `<input>`. Допустим, требуется добавить объект типа `FormControl`, чтобы предотвратить быструю передачу формы на обработку серверу, поскольку ее данные становятся достоверными только по истечении 10 секунд. С этой целью можно создать вручную объект типа `FormControl`, который будет иметь отношение к проверке достоверности комбинированной формы, но он не будет связан с элементом модели DOM.

См. также

- Рецепт "Реализация простой двунаправленной привязки данных с помощью директивы `ngModel`", демонстрирующий новый способ управления двунаправленным потоком данных в Angular 2.
- Рецепт "Объединение объектов типа `FormControl` средствами класса `FormGroup`", в котором описано, как объединять объекты типа `FormControl`.
- Рецепт "Объединение объектов типа `FormControl` средствами класса `FormArray`", поясняющий, как обращаться с итерируемыми элементами формы.

Объединение объектов типа `FormControl` средствами класса `FormGroup`

Естественно, что формы существуют в приложениях для того, чтобы свести несколько экземпляров элементов ввода данных к единому поведению. К числу наиболее распространенных видов поведения относится оценка

достоверности формы, для чего, безусловно, требуется, чтобы достоверными были данные во всех полях формы. И чаще всего это достигается объединением нескольких объектов типа `FormControl` в объект типа `FormGroup`. Сделать это можно несколькими способами с разной степенью явности. В данном рецепте рассматривается совершенно явная реализация, где все предполагается создавать и объединять вручную.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/3052/**.

Подготовка

Допустим, разработка начата с приведенной ниже заготовки. Цель состоит в том, чтобы обновить объект `article`, а следовательно, и шаблон, только в том случае, если достоверны данные, введенные во всех полях формы.

```
[app/article-editor.component.ts]

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

@Component({
  selector: 'article-editor',
  template: `
    <p>Title: <input></p>
    <p>Text: <input></p>
    <p><button (click)="saveArticle()">Save</button></p>
    <hr />
    <p>Preview:</p>
    <div style="border:1px solid #999;margin:50px;">
      <h1>{{article.title}}</h1>
      <p>{{article.text}}</p>
    </div>
  `
})
export class ArticleEditorComponent {
  article:{title:string, text:string} = {};
  saveArticle():void {}
}
```

Реализация

Прежде всего наберите исходный код, необходимый для присоединения новых объектов типа `FormControl` к каждому полю ввода данных и проверки их достоверности с помощью встроенного средства `required`:

```
[app/article-editor.component.ts]

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

```
import {FormControl, Validators}
        from '@angular/forms';

@Component({
  selector: 'article-editor',
  template: `
    <p>Title: <input [formControl]="titleControl"></p>
    <p>Text: <input [formControl]="textControl"></p>
    <p><button (click)="saveArticle()">Save</button></p>
    <hr />
    <p>Preview:</p>
    <div style="border:1px solid #999;margin:50px;">
      <h1>{{article.title}}</h1>
      <p>{{article.text}}</p>
    </div>
  `
})
export class ArticleEditorComponent {
  article:{title:string, text:string} = {};
  titleControl:FormControl
       = new FormControl(null, Validators.required);
  textControl:FormControl
       = new FormControl(null, Validators.required);

saveArticle():void {}
}
```

В данный момент можно было бы по отдельности исследовать объект типа FormControl для каждого элемента ввода данных и проверить его достоверность. Но если эта форма разрастется до 100 полей, то она станет невыносимо трудной для сопровождения. Но вместо этого объекты типа FormControl могут быть объединены в единый объект типа FormGroup следующим образом:

```
[app/article-editor.component.ts]

import {Component} from '@angular/core';
import {FormControl, FormGroup, Validators}
        from '@angular/forms';

@Component({
  selector: 'article-editor',
  template: `
    <p>Title: <input [formControl]="titleControl"></p>
    <p>Text: <input [formControl]="textControl"></p>
    <p><button (click)="saveArticle()">Save</button></p>
    <hr />
    <p>Preview:</p>
    <div style="border:1px solid #999;margin:50px;">
      <h1>{{article.title}}</h1>
```

```
      <p>{{article.text}}</p>
    </div>
  `
})
export class ArticleEditorComponent {
  article:{title:string, text:string} = {};
  titleControl:FormControl
       = new FormControl(null, Validators.required);
  textControl:FormControl
       = new FormControl(null, Validators.required);
  articleFormGroup:FormGroup = new FormGroup({
    title: this.titleControl,
    text: this.textControl
  });

  saveArticle():void {}
}
```

Объекты типа `FormGroup` имеют также свойства `valid` и `value`, которые можно использовать для проверки и непосредственного присваивания из объекта, как выделено ниже полужирным шрифтом. Благодаря этому дополнению рассматриваемая здесь форма будет теперь действовать нормально.

```
[app/article-editor.component.ts]

import {Component} from '@angular/core';
import {FormControl, FormGroup, Validators}
          from '@angular/forms';

@Component({
  selector: 'article-editor',
  template: `
    <p>Title: <input [formControl]="titleControl"></p>
    <p>Text: <input [formControl]="textControl"></p>
    <p><button (click)="saveArticle()">Save</button></p>
    <hr />
    <p>Preview:</p>
    <div style="border:1px solid #999;margin:50px;">
      <h1>{{article.title}}</h1>
      <p>{{article.text}}</p>
    </div>
  `
})
export class ArticleEditorComponent {
  article:{title:string, text:string} = {};
  titleControl:FormControl
       = new FormControl(null, Validators.required);
  textControl:FormControl
       = new FormControl(null, Validators.required);
  articleFormGroup:FormGroup = new FormGroup({
    title: this.titleControl,
```

```
    text: this.textControl
  });

  saveArticle():void {
    if (this.articleFormGroup.valid) {
      this.article = this.articleFormGroup.value;
    } else {
      alert("Missing field(s)!");
    }
  }
}
```

Принцип действия

Оба класса, `FormControl` и `FormGroup`, унаследованы от абстрактного базового класса `AbstractControl`. Это означает, что оба эти класса имеют те же самые методы, унаследованные из базового класса, но класс `FormGroup` составляет композицию из объектов типа `AbstractControl`, что позволяет читать данные из его собственных членов. Как следует из приведенного выше примера кода, свойство `valid` действует подобно логической операции И над всеми порожденными объектами, а это означает, что каждый операнд в отдельности должен возвратить логическое значение `true`, чтобы и вся логическая операция в целом возвратила логическое значение `true`. Таким образом, свойство `valid` возвращает объект с такой же топологией, как и предоставленная при получении экземпляра типа `FormGroup`, но только со значением, а не с объектом типа `FormControl`.

Как и следовало ожидать, в объекте типа **`FormGroup`** предполагается использовать объект со свойствами, относящимися к типу **`AbstractControl`**, и поэтому допускается свободное вложение одного объекта типа **`FormGroup`** в другой.

Дополнение

Свойства объекта типа `FormControl`, содержащегося в объекте типа `FormGroup`, могут быть доступны через свойство `controls`. Символьная строка, используемая для обозначения свойств объекта типа `FormControl`, будь то после получения экземпляра типа `FormGroup` или с помощью метода `addControl()`, служит для его извлечения. В данном примере объект `text` типа `FormControl` может быть извлечен в компонентном методе по ссылке `this.articleCtrlGroup.controls.text`.

В документации по Angular, в частности, предупреждается, что модифицировать коллекцию, положенную в основу класса **`FormControl`**, нежелательно, поскольку это может привести к неопределенному поведению при привязке данных. В связи с этим рекомендуется всегда пользоваться

компонентными методами **addControl()** и **removeControl()** из класса **FormGroup** вместо непосредственного манипулирования коллекцией объектов типа **FormControl**, которые передаются после получения экземпляров.

Средства проверки достоверности объектов типа FormGroup

Как и для объектов типа FormControl, для проверки достоверности объектов типа FormGroup имеются свои средства. Они могут быть предоставлены при получении экземпляра типа FormGroup и ведут себя таким же образом, как и средство проверки достоверности объекта типа FormControl. Вводя средства проверки достоверности на уровне класса FormGroup, можно переопределить в нем стандартное поведение, при котором объект оказывается достоверным лишь в том случае, если достоверны все его составляющие, а иначе вводятся дополнительные условия проверки достоверности.

Средства достоверности в Angular обладают возможностью не только определить достоверность данных, но и возвращать сообщения об ошибках, описывающие характер этих ошибок. Так, если проверить свойство errors объекта text типа FormControl, обратившись к нему по ссылке this.articleCtrlGroup.controls.text.errors, когда поля ввода данных оказываются пустыми, то из него возвратится значение {required: true}. Это стандартное сообщение об ошибке, возвращаемое встроенным средством проверки достоверности required. Но если проверить свойство errors в родительском объекте типа FormGroup по ссылке this.articleCtrlGroup.errors, то в нем окажется пустое значение null.

На первый взгляд это может показаться нелогичным, но на самом деле в этом нет никакой ошибки. Сообщения об ошибках появятся только в том экземпляре типа FormControl, в котором они произошли. Если же требуется объединить сообщения об ошибках, то придется обойти вручную вложенные коллекции объектов типа FormControl.

См. также

- Рецепт "Реализация простой двунаправленной привязки данных с помощью директивы ngModel", демонстрирующий новый способ управления двунаправленным потоком данных в Angular 2.
- Рецепт "Реализация элементарной проверки достоверности данных в поле средствами класса FormControl", подробно описывающий элементарный блок привязки в форме Angular.
- Рецепт "Объединение объектов типа FormControl средствами класса FormArray", в котором описано, как обращаться с итерируемыми элементами формы.

Объединение объектов типа FormControl средствами класса FormArray

Вы, вероятнее всего, обнаружите, что объекты типа FormGroup пригодны не только для объединения многих объектов типа FormControl в одном контейнере. Но имеется весьма распространенный шаблон, делающий чрезвычайно полезным сродный тип FormArray, представляющий клонированные вводимые данные переменной длины.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/2816/**.

Подготовка

Допустим, разработка начата с приведенной ниже заготовки. Цель состоит в том, чтобы добавить произвольное количество дескрипторов и проверить достоверность всех дескрипторов вместе.

```
[app/article-editor.component.ts]

import {Component} from '@angular/core';
import {FormControl, Validators}
        from '@angular/forms';

@Component({
  selector: 'article-editor',
  template: `
    <p>Tags:</p>
    <ul>
      <li *ngFor="let t of tagControls; let i = index">
         <input [formControl]="t">
      </li>
    </ul>
    <p><button (click)="addTag()">+</button></p>
    <p><button (click)="saveArticle()">Save</button></p>
  `
})
export class ArticleEditorComponent {
  tagControls:Array<FormControl> = [];
  addTag():void {}
  saveArticle():void {}
}
```

Реализация

Класс FormArray действует во многом так же, как и класс FormGroup. Он импортируется аналогично и наследуется из класса AbstractControl. А его

экземпляры получаются сходным образом и допускают ввод и удаление экземпляров типа FormControl. Прежде всего добавьте в свое приложение обычный код для получения экземпляра типа FormArray и передачи ему массива объектов типа FormControl, уже находящихся в компоненте. А поскольку уже имеется кнопка, предназначенная для вызова метода addTag(), то этот метод следует настроить на перенос объекта типа FormControl в массив tagControls, как выделено ниже полужирным шрифтом.

```
[app/article-editor.component.ts]

import {Component} from '@angular/core';
import {FormControl, FormArray, Validators}
        from '@angular/forms';

@Component({
  selector: 'article-editor',
  template: `
    <p>Tags:</p>
    <ul>
      <li *ngFor="let t of tagControls; let i = index">
        <input [formControl]="t">
      </li>
    </ul>
    <p><button (click)="addTag()">+</button></p>
    <p><button (click)="saveArticle()">Save</button></p>
  `
})
export class ArticleEditorComponent {
  tagControls:Array<FormControl> = [];
  tagFormArray:FormArray = new FormArray(this.tagControls);
  addTag():void {
    this.tagFormArray
        .push(new FormControl(null, Validators.required));
  }
  saveArticle():void {}
}
```

На данном этапе важно разобраться, с чем здесь приходится иметь дело. В компоненте **ArticleEditor** имеется массив **tagControls** объектов типа **FormControl**, а также единственный экземпляр типа **FormArray** (**tagFormArray**). При инициализации этого экземпляра конструктору класса **FormArray** передается массив объектов типа **FormControl**, чтобы в дальнейшем манипулировать им.

А теперь, когда объект типа FormArray может манипулировать объектами типа FormControl в дескрипторе разметки, можно благополучно воспользоваться его средством проверки достоверности следующим образом:

```
[app/article-editor.component.ts]

import {Component} from '@angular/core';
import {FormControl, FormArray, Validators}
        from '@angular/forms';

@Component({
  selector: 'article-editor',
  template: `
    <p>Tags:</p>
    <ul>
      <li *ngFor="let t of tagControls; let i = index">
        <input [formControl]="t">
      </li>
    </ul>
    <p><button (click)="addTag()">+</button></p>
    <p><button (click)="saveArticle()">Save</button></p>
  `
})
export class ArticleEditorComponent {
  tagControls:Array<FormControl> = [];
  tagFormArray:FormArray = new FormArray(this.tagControls);
  addTag():void {
    this.tagFormArray
        .push(new FormControl(null, Validators.required));
  }
  saveArticle():void {
    if (this.tagFormArray.valid) {
      alert('Valid!');
    } else {
      alert('Missing field(s)!');
    }
  }
}
```

Принцип действия

Шаблон реагирует на событие `click`, и поэтому для автоматического обновления этого шаблона можно воспользоваться доступной в Angular привязкой данных. Но в данном примере очень важно обратить внимание на присущую ему асимметрию. В шаблоне организуется обход массива `tagControls`. Но если требуется добавить новый объект типа `FormControl`, то он вносится сначала в массив `tagFormArray`, а оттуда — в массив `tagControls`. Объект типа `FormArray` действует в качестве диспетчера коллекции объектов типа `FormControl`, и все модификации этой коллекции должны проходить через ее диспетчер, а не через саму коллекцию.

В документации по Angular, в частности, предупреждается, что модифицировать коллекцию, положенную в основу класса **FormControl**, нежелательно, поскольку это может привести к неопределенному поведению при привязке данных. В связи с этим рекомендуется всегда пользоваться компонентными методами **push()**, **insert()** и **removeAt()** из класса **FormArray** вместо непосредственного манипулирования массивом объектов типа **FormControl**, которые передаются после получения экземпляров.

Дополнение

Данный пример можно расширить еще дальше, добавив возможность удалять объекты из данной коллекции. Реализовать такую возможность нетрудно, поскольку в повторителе шаблона уже имеется индекс, а в классе FormArray поддерживается удаление элементов массива по индексу. Ниже показано, как это делается. Такая возможность позволит ясно добавлять и удалять экземпляры типа FormControl, а все остальное автоматически сделает привязка данных в Angular.

```
[app/article-editor.component.ts]

import {Component} from '@angular/core';
import {FormControl, FormArray, Validators}
        from '@angular/forms';

@Component({
  selector: 'article-editor',
  template: `
    <p>Tags:</p>
    <ul>
      <li *ngFor="let t of tagControls; let i = index">
        <input [formControl]="t">
        <button (click)="removeTag(i)">X</button>
      </li>
    </ul>
     <p><button (click)="addTag()">+</button></p>
     <p><button (click)="saveArticle()">Save</button></p>
  `
})
export class ArticleEditorComponent {
  tagControls:Array<FormControl> = [];
  tagFormArray:FormArray = new FormArray(this.tagControls);
  addTag():void {
    this.tagFormArray
        .push(new FormControl(null, Validators.required));
  }
  removeTag(idx:number):void {
    this.tagFormArray.removeAt(idx);
  }
```

```
  saveArticle():void {
    if (this.tagFormArray.valid) {
      alert('Valid!');
    } else {
      alert('Missing field(s)!');
    }
  }
}
```

См. также

- Рецепт "Реализация простой двунаправленной привязки данных с помощью директивы ngModel", демонстрирующий новый способ управления двунаправленным потоком данных в Angular 2.
- Рецепт "Реализация элементарных форм с помощью директивы ngForm", предоставляющий декларативный способ создания формы в Angular.
- Рецепт "Реализация элементарных форм средствами класса FormBuilder и директивы formControlName", в котором поясняется, как пользоваться службой FormBuilder для быстрого составления вложенных форм.

Реализация элементарных форм с помощью директивы ngForm

Основные разновидности элементарных форм в Angular представлены объектами типа FormControl, FormGroup и FormArray. Но зачастую этими объектами можно вообще не пользоваться непосредственно. В каркасе Angular предоставляются механизмы, с помощью которых можно неявным образом создавать, присваивать и присоединять эти объекты к элементам модели DOM отдельной формы.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/5116/**.

Подготовка

Допустим, разработка начата с приведенной ниже заготовки. Цель состоит в том, чтобы собрать все данные в форме и передать их на обработку, используя конструкции форм, доступные в Angular.

```
[app/article-editor.component.ts]

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

```
@Component({
  selector: 'article-editor',
  template: `
    <p><input placeholder="Article title"></p>
    <p><textarea placeholder="Article text"></textarea></p>
    <p><button (click)="saveArticle()">Save</button></p>
  `
})
export class ArticleEditorComponent {
  saveArticle():void {}
}
```

Реализация

Прежде всего следует реорганизовать приведенный выше код в конкретную браузерную форму. Для этой цели в Angular предоставляется немало директив и компонентов. Импортировав модуль `FormsModule`, как выделено ниже полужирным шрифтом, можно получить постоянный доступ ко всем этим директивам.

```
[app/app.module.ts]

import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {FormsModule} from '@angular/forms';
import {ArticleEditorComponent} from './article-editor.component';

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

Кроме того, следует изменить конфигурацию экранной кнопки таким образом, чтобы превратить ее в кнопку `submit`. Когда эта кнопка нажимается для передачи данных формы на обработку серверу, должен запускаться соответствующий обработчик событий. Следовательно, приемник событий необходимо теперь присоединить к естественному событию `submit` формы, а не к событию `click` экранной кнопки. Наряду с этим событием в Angular предоставляется событие в виде свойства `ngSubmit` отправителя событий типа

EventEmitter, поэтому присоедините к нему приемник событий, как выделено ниже полужирным шрифтом.

[app/article-editor.component.ts]

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

@Component({
  selector: 'article-editor',
  template: `
    <form (ngSubmit)="saveArticle()">
      <p><input placeholder="Article title"></p>
      <p><textarea placeholder="Article text"></textarea></p>
      <p><button type="submit">Save</button></p>
    </form>
  `
})
export class ArticleEditorComponent {
  saveArticle():void {}
}
```

Далее следует настроить форму на передачу ее данных обработчику через переменную шаблона.

Элемент разметки формы будет содержать объект типа **NgForm** (а в нем — объект типа **FormGroup**), автоматически связываемый с ней при импорте модуля **FormsModule** в объемлющий модуль. Экземпляр объекта типа **NgForm** создается и связывается с формой в Angular автоматически.

Чтобы получить доступ к этому экземпляру, можно, в частности, присвоить директиву ngForm переменной шаблона. И хотя это похоже на некоторое синтаксическое волшебство, присваивание #f="ngForm" означает для Angular, что вам требуется возможность обращаться к объекту типа NgForm в самой форме из шаблона, используя переменную f.

Таким образом, объявив переменную шаблона, вы сможете теперь передать экземпляр директивы ngForm обработчику событий, наступающих при отправке формы на обработку, в качестве аргумента: saveArticle(f). В конечном итоге у вас должно получиться следующее:

[app/article-editor.component.ts]

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

@Component({
  selector: 'article-editor',
  template: `
    <form #f="ngForm"
          (ngSubmit)="saveArticle(f)">
```

```
      <p><input placeholder="Article title"></p>
      <p><textarea placeholder="Article text"></textarea></p>
      <p><button type="submit">Save</button></p>
    </form>
  `
})
export class ArticleEditorComponent {
  saveArticle(f:NgForm):void {
    console.log(f);
  }
}
```

Протестировав этот код вручную, вы должны увидеть, что браузер выводит на консоль объект типа `NgForm` всякий раз, когда вы щелкаете на экранной кнопке Save (Сохранить). А в самом объекте типа `NgForm` вы должны увидеть совершенно новый объект типа `FormGroup` и свойство `ngSubmit` отправителя событий типа `EventEmitter` как принимаемое событие. Это означает, что все пока что идет нормально!

Объявление полей формы с помощью директивы `ngModel`

Вы, вероятно, обратили внимание на то, что данные ни в одном из полей формы не были собраны. Это объясняется тем, что Angular не было предписано обратить на них внимание. Для сбора данных из полей формы в модуле `FormsModule` предоставляется директива `ngModel`, выполняющая следующие действия.

- Получение экземпляра объекта типа `FormControl`.
- Присоединение этого экземпляра к элементу модели DOM, внедряющему директиву `ngModel` в виде атрибута.
- Обнаружение объекта типа `FormGroup`, в котором находится данный элемент, и его добавление в только что созданный объект типа `FormControl`. Строковое значение в атрибуте `name` послужит ключом для обращения к нему в объекте типа `FormGroup`.

Последнее действие имеет особое значение, поскольку любая попытка воспользоваться директивой **`ngModel`** без присоединения объемлющей конструкции элемента управления формы приведет к ошибкам. В качестве такой конструкции в форме может служить сам объект типа **`FormGroup`** или даже экземпляр производного от него типа.

Итак, добавьте директиву `ngModel` в каждое поле ввода текста, как выделено ниже полужирным шрифтом.

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

```
@Component({
  selector: 'article-editor',
  template: `
    <form #f="ngForm"
          (ngSubmit)="saveArticle(f)">
      <p><input ngModel
                name="title"
                placeholder="Article title"></p>
      <p><textarea ngModel
                   name="text"
                   placeholder="Article text"></textarea></p>
      <p><button type="submit">Save</button></p>
    </form>
  `
})
export class ArticleEditorComponent {
  saveArticle(f:NgForm):void {
    console.log(f);
  }
}
```

Теперь ваша форма должна полностью функционировать. В обработчике событий, наступающих при отправке формы на обработку, можно проверить, что к объекту типа `FormGroup` присоединено два объекта типа `FormControl`. Для этого достаточно обратиться по ссылке `f.form.controls`, получив в итоге следующее:

```
{
  text: FormControl { ... },
  title: FormControl { ... }
}
```

Принцип действия

В данном рецепте, по существу, используется иерархический характер модели DOM, чтобы направить в нужное русло структурирование архитектуры объекта типа `FormControl`. В частности, самый верхний экземпляр типа `NgForm` связан с объектом типа `FormGroup`, где должны расположиться все остальные объекты типа `FormControl` данной формы.

Каждая директива `ngModel` направляет объект типа `FormControl`, на который она ссылается, объекту типа `FormGroup`, которым владеет экземпляр типа `NgForm`. Составив подобным образом вложенную структуру, можно прочитать из объекта типа `NgForm` и проанализировать состояние всей формы. В частности, передавая этот объект обработчику событий, наступающих при отправке формы на обработку, можно управлять каждым аспектом проверки состояния и достоверности формы.

Дополнение

Если же требуется сгруппировать некоторые поля формы, для этого достаточно заключить их в оболочку с помощью директивы ngModelGroup. Подобно директиве ngModel, она автоматически создает экземпляр типа FormGroup и присоединяет его к родительскому объекту типа FormGroup, а также добавляет любые заключенные в него объекты типа FormControl или FormGroup. Ниже приведен характерный тому пример.

```
[app/article.component.ts]

import {Component} from '@angular/core';
import {NgForm} from '@angular/forms';

@Component({
  selector: 'article-editor',
  template: `
    <form #f="ngForm"
          (ngSubmit)="saveArticle(f)">
      <div ngModelGroup="article">
        <p><input ngModel
            name="title"
            placeholder="Article title"></p>
        <p><textarea ngModel
            name="text"
            placeholder="Article text"></textarea></p>
      </div>
      <p><button type="submit">Save</button></p>
    </form>
  `
})
export class ArticleEditorComponent {
  saveArticle(f:NgForm):void {
    console.log(f);
  }
}
```

Если теперь провести инспекцию ссылки f.form.controls, то можно обнаружить, что в форме имеется единственный объект типа FormGroup, обозначенный ключом article, как показано ниже. Эта проверка проходит, поскольку данная структура совпадает с установленной в шаблоне.

```
{
  article: FormGroup: {
    controls: {
      text: FormControl { ... },
      title: FormControl { ... }
    },
    ...
  }
}
```

См. также

- Рецепт "Реализация простой двунаправленной привязки данных с помощью директивы ngModel", демонстрирующий новый способ управления двунаправленным потоком данных в Angular 2.
- Рецепт "Реализация элементарных форм средствами класса FormBuilder и директивы formControlName", в котором поясняется, как пользоваться службой типа FormBuilder для быстрого составления вложенных форм.

Реализация элементарных форм средствами класса FormBuilder и директивы formControlName

В каркасе Angular предоставляется готовый способ составления форм, определение которых не опирается на иерархию шаблонов. Вместо этого можно воспользоваться классом FormBuilder, чтобы сначала определить явным образом порядок структурирования объектов в форме, а затем присоединить их вручную к каждому элементу ввода данных.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/9302/**.

Подготовка

Допустим, разработка начата с приведенной ниже заготовки. Цель состоит в том, чтобы собрать все данные в форме и передать их на обработку, используя конструкции форм, доступные в Angular.

```
[app/article-editor.component.ts]

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

@Component({
  selector: 'article-editor',
  template: `
    <p><input placeholder="Article title"></p>
    <p><textarea placeholder="Article text"></textarea></p>
    <p><button (click)="saveArticle()">Save</button></p>
  `
})
export class ArticleEditorComponent {
  constructor() {}
  saveArticle():void {}
}
```

Реализация

Класс FormBuilder входит в состав модуля ReactiveFormsModule, поэтому в модуль приложения необходимо импортировать целевые элементы, выделенные ниже полужирным шрифтом.

[app/app.module.ts]

```
import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {ReactiveFormsModule} from '@angular/forms';
import {ArticleEditorComponent} from './article-editor.component';

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

Чтобы воспользоваться этим классом, внедрите его также в свой компонент. В каркасе Angular 2 для этого достаточно перечислить его в виде типизированного параметра конструктора. В классе FormBuilder предоставляется метод group(), возвращающий объект типа FormGroup верхнего уровня, который следует присвоить экземпляру вашего компонента. Пока что вам нужно передать пустой объект в качестве единственного аргумента данного метода.

Все сказанное выше означает, что вы можете внедрить объект articleGroup типа FormGroup в шаблон, присоединив его в дескрипторе form с помощью директивы formGroup следующим образом:

[app/article-editor.component.ts]

```
import {Component, Inject} from '@angular/core';
import {FormBuilder, FormGroup} from '@angular/forms';

@Component({
  selector: 'article-editor',
  template: `
    <form [formGroup]="articleGroup"
          (ngSubmit)="saveArticle()">
      <p><input placeholder="Article title"></p>
      <p><textarea placeholder="Article text"></textarea></p>
```

```
      <p><button type="submit">Save</button></p>
    </form>
  `
})
export class ArticleEditorComponent {
  articleGroup:FormGroup;
  constructor(@Inject(FormBuilder) formBuilder:FormBuilder) {
    this.articleGroup = formBuilder.group({});
  }
  saveArticle():void {}
}
```

Таким образом, вы успешно создали структуру для своей формы. Тем не менее объект типа `FormGroup` все еще не связан с несколькими полями для ввода данных. Для этого необходимо сначала установить структуру элементов управления в построителе формы и последовательно присоединить их к каждому дескриптору `input` с помощью директивы `formControlName` следующим образом:

```
[app/article-editor.component.ts]

import {Component, Inject} from '@angular/core';
import {FormBuilder, FormGroup, Validators} from '@angular/forms';

@Component({
  selector: 'article-editor',
  template: `
    <form [formGroup]="articleGroup"
          (ngSubmit)="saveArticle()">
      <p><input formControlName="title"
                placeholder="Article title"></p>
      <p><textarea formControlName="text"
                   placeholder="Article text"></textarea></p>
      <p><button type="submit">Save</button></p>
    </form>
  `
})
export class ArticleEditorComponent {
  articleGroup:FormGroup;

  constructor(@Inject(FormBuilder) formBuilder:FormBuilder) {
    this.articleGroup = formBuilder.group({
      title: [null, Validators.required],
      text: [null, Validators.required]
    });
  }
  saveArticle():void {
    console.log(this.articleGroup);
  }
}
```

В итоге ваша форма будет состоять из двух объектов типа FormControl, экземпляры которых получены в ней, и они будут связаны с соответствующими элементами разметки input. Когда вы щелкаете на экранной кнопке Submit (Передать на обработку), вы должны увидеть объекты типа FormControl, представляющие элементы ввода данных в объекте типа FormGroup. Но вы можете расположить эти объекты в обозначении свойства article, и для этого достаточно внести директиву ngFormGroup и соответствующий уровень косвенности в определение объекта formBuilder:

```
[app/article-editor.component.ts]

import {Component, Inject} from '@angular/core';
import {FormBuilder, FormGroup, Validators} from '@angular/forms';

@Component({
  selector: 'article-editor',
  template: `
    <form [formGroup]="articleGroup"
          (ngSubmit)="saveArticle()">
      <div formGroupName="article">
        <p><input formControlName="title"
                  placeholder="Article title"></p>
        <p><textarea formControlName="text"
                     placeholder="Article text"></textarea></p>
      </div>
      <p><button type="submit">Save</button></p>
    </form>
  `
})
export class ArticleEditorComponent {
  articleGroup:FormGroup;
  constructor(@Inject(FormBuilder) formBuilder:FormBuilder) {
    this.articleGroup = formBuilder.group({
      article: formBuilder.group({
        title: [null, Validators.required],
        text: [null, Validators.required]
      })
    });
  }
  saveArticle():void {
    console.log(this.articleGroup);
  }
}
```

Таким образом, свойства title и text будут существовать как объекты типа FormControl, вложенные в объект article типа FormGroup. Их можно успешно проверить на достоверность и состояние в обработчике событий, наступающих при отправке формы на обработку.

Принцип действия

Как можно предположить, массивы, существующие в определениях, сделанных в методе `formBuilder.group()`, могут быть указаны в качестве аргументов конструктора класса `FormControl`. И это замечательно, поскольку позволяет избежать стереотипной конструкции `new FormControl()` при создании каждого элемента управления. Символьная строка, служащая ключом для объекта типа `FormControl`, связывается с ним с помощью директивы `formControlName`. А поскольку в данном рецепте применяются директивы `formControlName` и `formGroupName`, то вложенную структуру `formBuilder` необходимо *точно* согласовать с тем, что находится в шаблоне.

Дополнение

Совершенно ясно, что необходимость дублировать структуру в определении шаблона и построителе форм типа `FormBuilder` несколько удручает. И это особенно справедливо в данном случае, поскольку наличие объекта `formGroup` совсем не предполагает добавление какого-нибудь ценного поведения из-за того, что они присоединены к инертному элементу разметки `div`. Вместо этого было бы желательно осуществить подобную группировку в пространстве имен статьи без модификации шаблона. И такого поведения можно добиться с помощью директивы `formControl`, действующей подобно директиве `formModel`. В частности, она производит привязку к экземпляру, существующему в компоненте, как показано ниже.

Обратите внимание на парадигму, которую демонстрируют упомянутые выше различные виды директив для составления форм. С помощью таких директив, как **`ngForm`**, **`formGroup`**, **`formArray`** и **`formControl`**, в Angular неявно создаются и связываются экземпляры объектов, составляющих форму. Если вы решите не пользоваться классом **`FormBuilder`** для определения конкретного поведения объектов типа **`FormControl`**, то можете добиться желаемого результата, введя директивы проверки достоверности в шаблон. А с другой стороны, можете воспользоваться директивами **`formModel`** и **`formControl`**, производящими привязку к экземплярам объектов, которые приходится создавать в компоненте вручную для каждого из элементов управления формы.

```
[app/article-editor.component.ts]

import {Component, Inject} from '@angular/core';
import {FormBuilder, FormControl, FormGroup, Validators}
        from '@angular/forms';

@Component({
  selector: 'article-editor',
  template: `
```

```
    <form [formGroup]="articleGroup"
          (ngSubmit)="saveArticle()">
      <p><input [formControl]="titleControl"
                placeholder="Article title"></p>
      <p><textarea [formControl]="textControl"
                   placeholder="Article text"></textarea></p>
      <p><button type="submit">Save</button></p>
    </form>
  `
})
export class ArticleEditorComponent {
  titleControl:FormControl
       = new FormControl(null, Validators.required);
  textControl:FormControl
       = new FormControl(null, Validators.required);
  articleGroup:FormGroup;
  constructor(@Inject(FormBuilder) formBuilder:FormBuilder) {
    this.articleGroup = formBuilder.group({
      article: formBuilder.group({
        title: this.titleControl,
        text: this.textControl
      })
    });
  }
  saveArticle():void {
    console.log(this.articleGroup);
  }
}
```

Важно отметить, что в данном примере был получен такой же результат, как и прежде. Объекты `title` и `text` объединены вместе в объекте `article` типа `FormGroup`. Но в шаблоне совсем не обязательно иметь какую-либо ссылку на этот промежуточный объект типа `FormGroup`.

См. также

- Рецепт "Реализация простой двунаправленной привязки данных с помощью директивы `ngModel`", демонстрирующий новый способ управления двунаправленным потоком данных в Angular 2.
- Рецепт "Реализация элементарной проверки достоверности данных в поле средствами класса `FormControl`", в котором подробно описывается элементарный блок привязки в форме Angular.
- Рецепт "Реализация элементарных форм с помощью директивы `ngForm`", предоставляющий декларативный способ создания формы в Angular.

Создание и применение специального средства проверки достоверности

Простейшие встроенные в Angular средства проверки достоверности позволяют вам начать действовать. Но если в вашем приложении активно используются формы, то вы, без сомнения, дойдете до того момента, когда вам потребуется определить логику собственного средства проверки достоверности.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/8574/**.

Подготовка

Допустим, разработка начата с приведенной ниже заготовки. Цель состоит в том, чтобы ввести дополнительную проверку достоверности в дескрипторе `textarea`, где объем вводимого текста должен быть ограничен десятью словами. (Краткость — излюбленная цель редакторов.)

```
[app/article-editor.component.ts]

import {Component} from '@angular/core';
import {FormControl, Validators} from '@angular/forms';

@Component({
  selector: 'article-editor',
  template: `
    <h2>Psych Study on Humility Wins Major Award</h2>
    <textarea [formControl]="bodyControl"
              placeholder="Article text"></textarea>
    <p><button (click)="saveArticle()">Save</button></p>
  `
})
export class ArticleEditorComponent {
  articleBody:string = '';
  bodyControl:Control
      = new FormControl(null, Validators.required);

  saveArticle():void {
    if (this.bodyControl.valid) {
      alert('Valid!');
    } else {
      alert('Invalid!');
    }
  }
}
```

Реализация

Глядя на сигнатуру конструктора класса `AbstractControl`, можно обнаружить, что средство проверки достоверности обозначается аргументом `ValidatorFn`. Это средство может быть любой функцией, которой передается объект типа `AbstractControl` в качестве единственного аргумента и возвращающей объект, обозначаемый модификатором типа `string` для сообщения об ошибке. Этот объект ошибки действует в качестве словаря ошибок, а средство проверки достоверности может возвратить столько ошибок, сколько уместно. Значение в словарной статье может (и должно) содержать метаданные о причине ошибки. Если же специальное средство проверки достоверности не обнаружит ошибки, оно должно просто возвратить пустое значение `null`.

Чтобы реализовать специальное средство проверки, проще всего добавить компонентный метод, как выделено ниже полужирным шрифтом.

```
[app/article-editor.component.ts]

export class ArticleEditor {
  articleBody:string
  bodyCtrl:Control
  constructor() {
    this.articleBody = '';
    this.bodyCtrl = new Control('', Validators.required);
  }
  wordCtValidator(c:Control): {[key: string]: any} {
    let wordCt:number = (c.value.match(/\S+/g) || []).length;
    return wordCt <= 10 ?
        null :
        { 'maxwords': { 'limit':10, 'actual':wordCt } };
  }
saveArticle() {
  if (this.bodyCtrl.valid) {
    alert('Valid!');
    } else {
      alert('Invalid!');
    }
  }
}
```

Здесь применяется регулярное выражение для сопоставления с любыми символьными строками без пробелов, которые можно рассматривать как отдельные слова. Необходимо также инициализировать объект типа **`FormControl`** пустой символьной строкой, поскольку здесь применяется метод **`match()`** прототипа **`string`**. Данное регулярное выражение возвратит пустое значение **`null`** в отсутствие совпадения, и поэтому в нем предусмотрено резервное предложение **`|| []`**, всегда выдающее какой-нибудь объект, имеющий метод **`length()`**.

А теперь, когда метод проверки достоверности определен, необходимо применить его в объекте типа FormControl. В каркасе Angular допускается объединять массив средств проверки достоверности в единое средство, вычисляющее их по порядку, как выделено ниже полужирным шрифтом. Таким образом, объект типа FormControl должен оказаться достоверным лишь в том случае, если в нем введено не больше десяти слов и он не пуст.

```
[app/article-editor.component.ts]

import {Component} from '@angular/core';
import {FormControl, Validators} from '@angular/forms';

@Component({
  selector: 'article-editor',
  template: `
    <h2>Psych Study on Humility Wins Major Award</h2>
    <textarea [formControl]="bodyControl"
              placeholder="Article text"></textarea>
    <p><button (click)="saveArticle()">Save</button></p>
  `
})
export class ArticleEditorComponent {
  articleBody:string = '';
  bodyControl:FormControl = new FormControl(null,
      [Validators.required, this.wordCtValidator]);

  wordCtValidator(c:FormControl):{[key: string]: any} {
    let wordCt:number
        = ((c.value || '').match(/\S+/g) || []).length;
    return wordCt <= 10 ?
       null :
       {maxwords: {limit:10, actual:wordCt}};
  }

  saveArticle():void {
    if (this.bodyControl.valid) {
      alert('Valid!');
    } else {
      alert('Invalid!');
    }
  }
}
```

Принцип действия

В конструкторе класса FormControl предполагается аргумент ValidatorFn с указанным возвращаемым типом, но его происхождение не имеет значения.

Следовательно, это дает возможность определить метод в классе компонента и передать его при получении экземпляра типа `FormControl`.

Объект типа `FormControl`, связанный с некоторыми вводимыми данными, должен обладать соответствующими средствами проверки достоверности. Сначала в данном рецепте была реализована специальная проверка достоверности. С этой целью была установлена явная ассоциация через аргументы получения экземпляров и определение средства проверки достоверности в качестве самостоятельного аргумента `ValidatorFn`.

Дополнение

Такое решение совершенно не удовлетворит опытного разработчика программного обеспечения. Рассмотренное выше специальное средство проверки достоверности нельзя применять за пределами компонента, в котором оно было определено, не внедрив весь этот компонент. А самое неприятное, что при получении экземпляра типа `FormControl` приходится явно указывать каждое средство проверки достоверности.

Преобразование в атрибуты проверки достоверности

Более совершенное решение состоит в том, чтобы реализовать формальный класс типа `Validator`. Такое решение обладает тем преимуществом, что оно позволяет импортировать и экспортировать этот класс, используя средство проверки достоверности как атрибут в шаблоне. Благодаря этому отпадает необходимость объединять средства проверки достоверности с помощью метода `Validators.compose()`.

Цель состоит в том, чтобы создать директиву, способную функционировать не только как атрибут, но и как нечто такое, что Angular сможет распознать как формальное средство проверки достоверности типа `Validator` и автоматически внедрить его. Для достижения этой цели можно создать директиву, реализующую интерфейс `Validator`, связав новую директиву с маркером `NG_VALIDATORS`.

Не обращайте пока что особого внимания на то, что происходит с массивом **`providers`** в объекте метаданных директивы. Более подробно об этом речь пойдет в главе 7, посвященной внедрению зависимостей. А до тех пор достаточно знать, что в рассматриваемом здесь коде допускается привязывать объект типа **`FormControl`** к дескриптору **`textarea`**, чтобы связать с ним создаваемое средство проверки достоверности.

Прежде всего перенесите метод проверки достоверности в отдельную директиву, выполнив указанные выше действия, как выделено ниже полужирным шрифтом.

[app/max-word-count.validator.ts]

```
import {Directive} from '@angular/core';
import {Validator, FormControl, NG_VALIDATORS}
       from '@angular/forms';

@Directive({
  selector: '[max-word-count]',
  providers: [{
    provide:NG_VALIDATORS,
    useExisting: MaxWordCountValidator,
    multi: true
  }]
})
export class MaxWordCountValidator implements Validator {
  validate(c:FormControl):{[key:string]:any} {
    let wordCt:number = ((c.value || '')
              .match(/\S+/g) || []).length;
    return wordCt <= 10 ?
           null :
           {maxwords: {limit:10, actual:wordCt}};
  }
}
```

Далее добавьте следующую директиву в модуль приложения:

[app/app.module.ts]

```
import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {ReactiveFormsModule} from '@angular/forms';
import {ArticleEditorComponent} from './article-editor.component';
import {MaxWordCountValidator} from './max-word-count.validator';

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

В итоге эта директива станет доступной всем компонентам в данном модуле. Более того, указанная прежде конфигурация поставщика позволяет просто добавить атрибут директивы в любой элемент ввода данных, и тогда Angular внедрит его метод проверки достоверности в соответствующий объект типа `FormControl`. Подобная интеграция осуществляется следующим образом:

```
[app/article-editor.component.ts]

import {Component} from '@angular/core';
import {FormControl} from '@angular/forms';

@Component({
  selector: 'article-editor',
  template: `
    <h2>Psych Study on Humility Wins Major Award</h2>
    <textarea [formControl]="bodyControl"
              required
              max-word-count
              placeholder="Article text"></textarea>
    <p><button (click)="saveArticle()">Save</button></p>
  `
})
export class ArticleEditorComponent {
  articleBody:string = '';
  bodyControl:FormControl = new FormControl();

  saveArticle():void {
    if (this.bodyControl.valid) {
      alert('Valid!');
    } else {
      alert('Invalid!');
    }
  }
}
```

Уже намного лучше. Теперь директиву `max-word-count` можно импортировать и применять повсюду в приложении, просто указав ее как зависимость от директивы в компоненте. А для получения экземпляра объекта типа `FormControl` больше не придется вызывать метод `Validator.compose()` со всеми сопутствующими неудобствами.

Это особенно удобно при неявном создании объектов типа **FormControl** с помощью директивы **formControl** и прочих встроенных директив для форм, что для большинства приложений окажется основным способом использования форм. В подобных случаях создание средства проверки достоверности в виде атрибута плавно впишется в общий процесс разработки.

Данное решение все же не вполне удовлетворительно, поскольку средство проверки достоверности жестко закодировано на проверку 10 слов. Это лучше было бы поручить элементу ввода текстовых данных, в котором применяется данное средство. Поэтому рассматриваемую здесь директиву необходимо изменить таким образом, чтобы ей передавался единственный параметр в форме значения атрибута, как выделено ниже полужирным шрифтом. Таким образом, значение атрибута определено в качестве входных данных для средства проверки достоверности и может быть далее использовано для настройки режима работы этого средства.

[app/max-word-count.validator.ts]

```
import {Directive} from '@angular/core';
import {Validator, FormControl, NG_VALIDATORS}
       from '@angular/forms';

@Directive({
  selector: '[max-word-count]',
  inputs: ['rawCount: max-word-count'],
  providers: [{
    provide:NG_VALIDATORS,
    useExisting: MaxWordCountValidator,
    multi: true
  }]
})
export class MaxWordCountValidator implements Validator {
  rawCount:string;
  validate(c:FormControl):{[key:string]:any} {
  let wordCt:number =
      ((c.value || '').match(/\S+/g) || []).length;
  return wordCt <= this.maxCount ?
      null :
      {maxwords: {limit:this.maxCount, actual:wordCt}};
  }
  get maxCount():number {
    return parseInt(this.rawCount);
  }
}
```

[app/article-editor.component.ts]

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

@Component({
  selector: 'article-editor',
  template: `
    <h2>Psych Study on Humility Wins Major Award</h2>
    <textarea [formControl]="bodyControl"
              required
```

```
            max-word-count="10"
            placeholder="Article text"></textarea>
    <p><button (click)="saveArticle()">Save</button></p>
  `
})
export class ArticleEditorComponent {
  articleBody:string = '';
  bodyControl:FormControl = new FormControl();
  saveArticle():void {
    if (this.bodyControl.valid) {
      alert('Valid!');
    } else {
      alert('Invalid!');
    }
  }
}
```

См. также

- Рецепт "Создание и применение асинхронного средства проверки достоверности с помощью обязательств", в котором показано, как в Angular организуется отложенное вычисление состояния формы.

Создание и применение асинхронного средства проверки достоверности с помощью обязательств

Стандартное средство проверки достоверности действует при условии, что достоверность некоторых введенных данных может быть определена за короткий период времени, в течение которого приложение готово подождать, прежде чем продолжить работу. Более того, Angular будет выполнять проверку достоверности всякий раз, когда для этого вызывается соответствующее средство. А это может происходить довольно часто, если проверка достоверности формы привязана к скоротечным событиям вроде нажатий клавиш.

Поэтому было бы неплохо, если бы существовала конструкция, позволяющая плавно выполнять процедуры проверки достоверности, отнимающие произвольное количество времени, или же процедуры, которые могут вообще ничего не возвращать. Для этой цели в Angular предоставляется *асинхронное* средство проверки достоверности, полностью совместимое с обязательствами.

Исходный код, ссыл5ки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/7811/**.

Подготовка

Допустим, что разработка начата с приведенной ниже заготовки. Цель состоит в настройке данной формы таким образом, чтобы она стала достоверной только через 5 секунд после того, как пользователь введет данные. Это нужно для противодействия спамботам (программам, автоматически собирающим адреса для рассылки спама — рекламных и прочих навязчивых сообщений).

```
[app/article-editor.component.ts]

import {Component} from '@angular/core';
import {FormControl} from '@angular/forms';

@Component({
  selector: 'article-editor',
  template: `
    <h2>New York-Style Pizza Actually Saucy Cardboard</h2>
    <textarea [formControl]="bodyControl"
              placeholder="Article text">
    </textarea>
    <p><button (click)="saveArticle()">Save</button></p>
  `
})
export class ArticleEditorComponent {
  articleBody:string = '';
  bodyControl:FormControl = new FormControl();
  saveArticle():void {
  if (this.bodyControl.valid) {
    alert('Valid!');
    } else {
      alert('Invalid!');
    }
  }
}
```

Реализация

Прежде всего создайте класс для средства проверки достоверности, а в нем — статический метод проверки достоверности. Он подобен синхронному методу проверки достоверности, но возвращает обязательство в виде объекта типа `Promise`, передавая результирующие данные методу `resolve()`, который разрешает данное обязательство. Объекту типа `FormControl` передается асинхронное средство проверки достоверности в качестве третьего аргумента. Если же обычные средства проверки достоверности не применяются, то данный аргумент можно оставить пустым (`null`).

Если обычные средства проверки достоверности объединяются вместе с помощью метода **Validators.compose()**, то асинхронные средства проверки достоверности можно объединять с помощью метода **Validators.composeAsync()**.

Создайте следующую заготовку средства проверки достоверности в отдельном файле:

```
[app/delay.validator.ts]

import {FormControl, Validator} from '@angular/forms';

export class DelayValidator implements Validator {
  static validate(c:FormControl):Promise<{[key:string]:any}> {
  }
}
```

И хотя это средство проверки достоверности пока еще ничего не делает, его можно добавить к компоненту, как выделено ниже полужирным шрифтом.

```
[app/article-editor.component.ts]

import {Component} from '@angular/core';
import {FormControl, Validators} from '@angular/forms';
import {DelayValidator} from './delay.validator';

@Component({
  selector: 'article-editor',
  template: `
    <h2>New York-Style Pizza Actually Saucy Cardboard</h2>
    <textarea [formControl]="bodyControl"
              placeholder="Article text">
    </textarea>
    <p><button (click)="saveArticle()">Save</button></p>
  `
})
export class ArticleEditorComponent {
  articleBody:string = '';
  bodyControl:FormControl =
    new FormControl(null, null, DelayValidator.validate);
  saveArticle():void {
    if (this.bodyControl.valid) {
      alert('Valid!');
    } else {
      alert('Invalid!');
    }
  }
}
```

Данное средство проверки достоверности должно возвращать обязательство, которое вообще не должно быть разрешено. Более того, задержку

желательно устанавливать лишь по одному разу на каждую визуализацию формы. Поэтому в данном случае обязательство можно просто присоединить к объекту типа FormControl, как показано ниже. Благодаря такому дополнению форма будет оставаться недостоверной в течение 5 секунд после первого изменения значения в дескрипторе textarea.

```
[app/delay.validator.ts]

import {FormControl, Validator} from '@angular/forms';

export class DelayValidator implements Validator {
  static validate(c:FormControl):Promise<{[key:string]:any}> {
    if (c.pristine && !c.value) {
      return new Promise;
    }
    if (!c.delayPromise) {
      c.delayPromise = new Promise((resolve) => {
        setTimeout(() => {
          console.log('resolve');
          resolve();
        }, 5000);
      });
    }
    return c.delayPromise;
  }
}
```

Принцип действия

Асинхронные средства проверки достоверности выполняются независимо через обычные (синхронные) средства проверки достоверности, но, помимо отличий во внутренних задержках, они в конечном счете ведут себя практически одинаково. Главное отличие заключается в том, что асинхронное средство проверки достоверности обладает состоянием ожидания, помимо достоверных и недостоверных состояний, общих с обычным средством проверки достоверности. Объект типа FormControl останется в этом состоянии до тех пор, пока не будет дано обязательство разрешить или отклонить результат, возвращаемый средством проверки достоверности.

Объект типа **FormControl**, находящийся в состоянии *ожидания*, считается недостоверным для целей проверки достоверности составных конструкций вроде **FormGroup** или **FormArray**.

В созданном только что средстве проверки достоверности достаточно проверить свойство pristine объекта типа FormControl, чтобы убедиться,

является ли форма "свежей" или нет. Прежде чем пользователь внесет изменения во введенные данные, свойство pristine примет логическое значение true, а после любого изменения (и даже удаления всех введенных данных) свойство pristine примет логическое значение false. Следовательно, такое поведение идеально подходит в данном случае, поскольку оно позволяет сохранять состояние формы в объекте типа FormControl, не усложняя чрезмерно средство проверки достоверности.

Дополнение

Особое внимание обращает на себя форма, которую принимает рассматриваемое здесь средство проверки достоверности. Метод проверки достоверности определен в классе DelayValidator как статический, а экземпляры этого класса нигде больше не получаются. Единственное назначение класса DelayValidator — служить вместилищем для средства проверки достоверности. Хранить информацию в данном классе нельзя, поскольку отсутствуют его экземпляры, где это можно было бы сделать.

В данном примере может возникнуть искушение добавить компонентные данные в средство проверки достоверности, поскольку требуется каким-то образом отслеживать изменения во вводимых данных. Но поступить так — значит следовать антишаблону! В данном случае объект типа **FormControl** должен действовать как единственный источник информации о внутреннем состоянии. Для каждого поля ввода данных получается экземпляр объекта типа **FormControl**, а следовательно, он служит идеальным "хранилищем", с помощью которого можно отслеживать состояние отдельного элемента ввода данных.

Выполнение средства проверки достоверности

Если попытаться выяснить, когда вызывается метод проверки достоверности, то можно обнаружить, что он выполняется только при нажатии клавиши в текстовой области, размечаемой дескриптором textarea. Такое поведение может показаться произвольным, но стандартное присваивание объектов типа FormControl элементам разметки input служит для того, чтобы средства проверки достоверности выполнялись в этих объектах при наступлении события change от ввода данных. Объекты типа FormControl предоставляют метод registerOnChange(), позволяющий осуществлять перехват в тот момент, когда будут выполняться средства проверки достоверности.

См. также

- Рецепт "Создание и применение специального средства проверки достоверности", в котором демонстрируется порядок создания специальной директивы, действующей подобно проверке достоверности.

Глава 4

Использование обязательств

В этой главе даются следующие рецепты.

- Представление и реализация основных обязательств
- Связывание обязательств и их обработчиков в цепочку
- Создание оболочек для обязательств с помощью методов `Promise.resolve()` и `Promise.reject()`
- Реализация барьеров для обязательств с помощью метода `Promise.all()`
- Отмена асинхронных действий с помощью метода `Promise.race()`
- Преобразование обязательств в наблюдаемые объекты
- Преобразование наблюдаемого объекта HTTP-службы в обязательство типа `ZoneAwarePromise`

Введение

В каркасе Angular 1 обязательства действовали как чужеродные элементы. Они были важны для написания надежных асинхронных приложений, но их применение, по-видимому, обходилось недаром. Их реализация посредством службы `$q` и двойственность объектов `promise` и `deferred` казались довольно необычными. Тем не менее, овладев ими, нетрудно было обнаружить, что они могли служить основанием для весьма надежных реализаций в однопоточной и управляемой событиями среде JavaScript.

К счастью для разработчиков, компонент обязательства `Promise` формально стал центральным в стандарте ES6 языка JavaScript. А поскольку язык TypeScript является надмножеством, расширяющим возможности стандарта ES6, то вам будет приятно узнать, что теперь вы можете пользоваться

обязательствами в Angular без дополнительной нагрузки. И хотя наблюдаемые объекты включают в себя немало функциональных возможностей, предоставляемых обязательствами, последним, тем не менее, найдется немало места в арсенале ваших инструментальных средств.

Возможность пользоваться обязательствами в исходном виде является привилегией траспиляциии из TypeScript в JavaScript. В настоящее время обязательства исходно поддерживаются в одних браузерах, но не поддерживаются в других. Но это и не так важно, если написать приложение на TypeScript и траспилировать его надлежащим образом! На самом деле механизм траспиляциии необходим лишь в том случае, если требуется воспользоваться преимуществами платформенно-ориентированных реализаций по сравнению с полифилами в отношении производительности и объема полезной нагрузки. И это не должно служить препятствием для разработки подавляющего большинства приложений.

Представление и реализация основных обязательств

Обязательства (promises) очень удобны во многих основных аспектах применения Angular. И хотя обязательства больше не привязаны к базовой службе каркаса Angular, они проявляются в его интерфейсах API. Они реализуются в Angular 2 значительно проще, чем в Angular 1, хотя основные стадии данного процесса остались прежними.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **`http://ngcookbook.herokuapp.com/5195/`**.

Подготовка

Прежде чем пользоваться обязательствами, необходимо уяснить затруднение, которое они призваны разрешить. Не особенно вдаваясь в подробности, принцип выполнения обязательства можно разделить на следующие стадии.

- **Инициализация.** Допустим, имеется работа, которую требуется завершить, но в то же время требуется определить, что должно произойти, когда она завершится. При этом неизвестно, когда работа завершится и произойдет ли это удачно или неудачно.
- **Ожидание.** Работа начата, но еще не завершена.
- **Завершение.** Работа завершена, и обязательства переходят в оконечное состояние. Это "завершенное" состояние предполагает две формы: "выполнено" и "отклонено". Эти формы соответствуют удачному и неудачному завершению соответственно.

Имеются и другие особенности работы обязательств, но и приведенных выше пояснений должно быть достаточно, чтобы перейти к написанию кода.

Реализация

Ниже приведена одна из простейших форм реализации обязательства.

```
// экземпляры обязательств получаются с помощью ключевого слова new
var promise = new Promise(() => {});
```

Функция, передаваемая конструктору класса `Promise`, представляет собой часть работы, которую предполагается выполнить асинхронно. Формально она обозначается как *исполнитель*.

Конструктор класса **`Promise`** вообще не интересует поведение функции-исполнителя. Он лишь предоставляет ей две функции: **`resolve()`** и **`reject()`**. А функции-исполнителю остается только вызвать их в нужный момент. Однако функция-исполнитель совсем не обязательно должна быть асинхронной. Но если она не является асинхронной, то и не требуется обязательство на то, что предполагается выполнить.

Когда функция-исполнитель выполняется, то ей изнутри известно, когда она завершится. Но снаружи отсутствует конструкция, представляющая следующий принцип: "выполнить обязательство по завершении функции-исполнителя". Следовательно, первыми двумя параметрами данной функции являются функции `resolve()` и `reject()`. Обязательство, заключающее функцию-исполнитель в оболочку, находится в состоянии "ожидание" до тех пор, пока не будет вызвана одна из этих функций. И как только она будет вызвана, обязательство необратимо перейдет в соответствующее состояние.

Функция-исполнитель вызывается сразу же после создания экземпляра обязательства. Не менее важно отметить, что она вызывается прежде, чем возвратится полученный экземпляр обязательства. Это означает, что если обязательство синхронно достигнет состояния "выполнено" или "отклонено" в теле функции-исполнителя, операция **`new Promise(...)`** возвратит вновь созданное обязательство, находящееся в состоянии "выполнено" или "отклонено". При этом стадия ожидания полностью пропускается.

Не имеет особого значения, что именно возвращает функция-исполнитель. Что бы она ни возвращала, конструктор класса **`Promise`** всегда возвратит вновь созданное обязательство.

В следующем примере кода демонстрируются пять способов получения экземпляра объекта типа `Promise`, выполнения и отклонения обязательства:

```
// Этому исполнителю передаются функции resolve() и reject(),
// но он, по существу, не выполняет никакой операции, и поэтому
// обязательство p1 навсегда остается в состоянии "ожидание".
```

```
const p1 = new Promise((resolve, reject) => {});

// Этот исполнитель сразу же вызывает функцию resolve(),
// и поэтому обязательство p2 перейдет непосредственно
// в состояние "выполнено".
const p2 = new Promise((resolve, reject) => resolve());

// Этот исполнитель сразу же вызывает функцию reject(),
// и поэтому обязательство p3 перейдет непосредственно
// в состояние "отклонено". Переход в состояние "отклонено"
// приведет также к генерации исключения по завершении исполнителя.
// Учтите, что любой код, следующий после вызова функции reject(),
// будет также выполнен.
const p3 = new Promise((resolve, reject) => {
  reject();
  // В следующей строке кода функция log() выведет
  // сообщение, прежде чем будет сгенерировано исключение
  console.log('Обязательство отклонено!');
});

// Этот исполнитель сразу же вызывает функцию resolve(),
// и поэтому обязательство p4 перейдет непосредственно
// в состояние "выполнено". Как только обязательство
// покинет состояние ожидания, его состояние уже нельзя
// будет изменить на что-то другое. И несмотря на то что в
// дальнейшем вызывается функция reject(), в конечном
// итоге обязательство p4 все равно окажется
// в состоянии "выполнено".
const p4 = new Promise((resolve, reject) => {
  resolve();
  reject();
});

// Этот исполнитель присваивает свою функцию resolve()
// переменной в объемлющей лексической области видимости,
// и поэтому его можно вызвать за пределами определения
// обязательства.
var outerResolve;
const p5 = new Promise((resolve, reject) => {
  outerResolve = resolve();
});
// Обязательство p5 находится в состоянии "ожидание"

outerResolve();
// Обязательство p5 находится в состоянии "выполнено"
```

Из приведенных выше примеров совсем не следует, что конструкции обязательства можно найти какое-нибудь полезное применение. Дело в том, что в коде из этих примеров лишь устанавливается состояние отдельного обязательства. А истинный потенциал обязательств раскрывается при

последующей установке обработчиков состояний. В интерфейсе API класса Promise предоставляется метод then(), позволяющий устанавливать обработчики состояний таким образом, чтобы они выполнялись, как только обязательство достигнет своего оконечного состояния, как показано ниже.

```
// p1 - это простое обязательство, к которому можно
// присоединить обработчики состояний
const p1 = new Promise((resolve, reject) => {});

// Обязательство p1 предоставляет метод then(),
// которому передается обработчик состояния "выполнено"
// (onFulfilled), а также обработчик состояния
// "отклонено" (onRejected)
p1.then(
  // Обработчик onFulfilled() вызывается при вызове
  // функции resolve()
  () => {},
  // Обработчик onRejected()вызывается при вызове
  // функции reject()
  () => {});
// Если оставить этот код без изменения,
// то обязательство p1 навсегда останется
// в состоянии ожидания

// С помощью ключевого слова new можно по-прежнему
// вызвать метод для возвращаемого экземпляра.
// Поэтому сразу же разрешается определить обработчики
// состояний в методе then().
//
// Немедленно выполнить обязательство p2
const p2 = new Promise((resolve, reject) => resolve())
  .then(
  // Этот метод вызывается сразу же после вызова
  // функции resolve() в исполнителе обязательства p2
  () => console.log('Выполнено!'));
  // Сообщение "Выполнено!"

// Немедленно отклонить обязательство p3
const p3 = new Promise((resolve, reject) => reject())
  .then(
  () => console.log('Выполнено!'),
  // Второй метод вызывается сразу же после вызова
  // функции reject() в исполнителе обязательства p3
  () => console.log('Отклонено!'));
  // Сообщение "Отклонено!"

const p4 = new Promise((resolve, reject) => reject())
  // Если не требуется обработчик состояния "выполнено",
  // обработку ошибок можно организовать с помощью
  // функции catch()
  .catch(() => console.log('Отклонено!'));
```

```
// Параметры исполнителя могут быть зафиксированы за
// пределами его лексической области видимости для
// последующего вызова
var outerResolve;
const p5 = new Promise((resolve, reject) => {
  outerResolve = resolve;
}).then(() => console.log('Выполнено!'));

outerResolve();
// Сообщение "Выполнено!"
```

Принцип действия

Обязательства в JavaScript упрощают разработчикам задачу написания асинхронного кода параллельно с синхронным кодом. В языке JavaScript эта задача раньше решалась с помощью вложенных функций обратного вызова, в просторечии называемых "адом обратных вызовов". Так, одиночную функцию, в которой используется обратный вызов, можно написать следующим образом:

```
// Типичная асинхронная функция обратного вызова
function asyncFunction(data, successCallback, errorCallback) {
  // Эта функция выполнит некоторую операцию, которая может
  // завершиться удачно, неудачно или не возвратить вообще
  // ничего, причем любой из этих исходов может наступить
  // в неизвестный заранее момент времени

  // Этот псевдответ содержит логическое значение,
  // обозначающее удачный исход, а также данные,
  // возвращаемые при удачном исходе операции
  asyncOperation(data, function(response) {
    if (response.success === true) {
      successCallback(response.data);
    } else {
      errorCallback();
    }
  });
};
```

Если приложение не требует какого-то подобия упорядоченного или общего завершения операций, то достаточно будет и следующего кода:

```
function successCallback(data) {
  // Функция asyncFunction() завершилась удачно,
  // обработать результат надлежащим образом
};

function errorCallback() {
  // Функция asyncFunction() завершилась неудачно,
  // обработать результат надлежащим образом
};
```

```
asyncFunction(data1, successCallback, errorCallback);
asyncFunction(data2, successCallback, errorCallback);
asyncFunction(data3, successCallback, errorCallback);
```

Но так почти никогда не бывает. Зачастую в приложении требуется, чтобы данные поступали в определенной последовательности или чтобы операция обработки данных была выполнена, как только будут успешно получены все данные, асинхронно поступающие отдельными фрагментами. И если обязательства недоступны, тогда и возникает "ад обратных вызовов", как показано ниже.

```
asyncFunction(data1, (foo) => {
  asyncFunction(data2, (bar) => {
    asyncFunction(data3, (baz) => {
      // Теперь все функции foo(), bar(), baz()
      // могут использоваться вместе
      combinatoricFunction(foo, bar, baz);
    }, errorCallback);
  }, errorCallback);
}, errorCallback);
```

Этот так называемый "ад обратных вызовов" возникает здесь из-за попытки выполнить последовательно три асинхронных вызова. Но параметрическая топология этих асинхронных функций вынуждает разработчика повергать свое приложение в подобную мерзость.

Дополнение

В отношении обязательств очень важно иметь в виду, что они позволяют разбить вычисление на две части: часть, где становится известно, когда "выполнение" обязательства завершилось, и часть, где оставшаяся программа извещается, что выполнение завершено.

Несвязное и дублированное управление обязательствами

Обязательство способно уступить контроль над тем, кто решает, где должно быть оно готово, и поэтому состояние обязательства может быть установлено во многих внешних частях кода.

Экземпляр обязательства может быть выполнен или отклонен в нескольких местах самого исполнителя:

```
const p = new Promise((resolve, reject) => {
  // Следующие псевдометоды могут быть вызваны независимо,
  // асинхронно или вообще не быть вызванными
  function canHappenFirst() { resolve(); };
  function mayHappenFirst() { resolve(); }
  function mightHappenFirst() { reject(); };
});
```

Экземпляр обязательства может быть также выполнен в нескольких местах за пределами исполнителя, как показано ниже.

```
var outerResolve;
const p = new Promise((resolve, reject) => {
  outerResolve = resolve;
});

// Следующие псевдометоды могут быть вызваны независимо,
// асинхронно или вообще не быть вызванными
function canHappenFirst() { outerResolve (); };
function mayHappenFirst() { outerResolve (); }
function mightHappenFirst() { outerResolve (); };
```

Как только обязательство перейдет в состояние "выполнено" или "отклонено", любые последующие попытки отклонить или разрешить это обязательство будут негласно проигнорированы. Обязательство переходит из одного состояния в другое только один раз и не может изменить текущее состояние или вернуться в прежнее.

Разрешение обязательства в значение

Принцип построения конструкций обязательств отчасти состоит в следующем: они обязуются предоставить соответствующее значение, которое станет доступным когда обязательство будет выполнено. С отдельными состояниями совсем не обязательно связано какое-то конкретное значение типа данных. Они лишь наделяют обязательство определенным состоянием вычисления, как показано ниже.

```
var resolveHandler = () => {},
    rejectHandler = () => {};
const p0 = new Promise((resolve, reject) => {
  // Состояние может быть определено одним из следующих способов:
  // resolve();
  // reject();
  // resolve(myData);
  // reject(myData);
}).then(resolveHandler, rejectHandler);
```

Вычисляемое (выполняемое или отклоняемое) обязательство связано с обработчиком каждого его состояния. Этот обработчик вызывается при переходе обязательства в соответствующее состояние. Обработчики состояний могут получать доступ к данным, возвращаемым операцией выполнения или отклонения обязательства, как демонстрируется в следующем примере кода:

```
const p1 = new Promise((resolve, reject) => {
  // console.info() – это обработчик состояния "выполнено",
  // console.error() - а это обработчик состояния "отклонено"
  resolve(123);
```

```
}).then(console.info, console.error);

// (info) 123

// установить в исходное состояние, чтобы
// продемонстрировать функцию reject()
const p2 = new Promise((resolve, reject) => {
  // console.info() — это обработчик состояния "выполнено",
  // console.error() - а это обработчик состояния "отклонено"
  reject(456);
}).then(console.info, console.error);

// (error) 456
```

Отложенное определение обработчиков

В отличие от функций обратного вызова, обработчики могут быть определены в любой момент жизненного цикла обязательства, в том числе и после того, как будет определено состояние обязательства:

```
const p3 = new Promise((resolve, reject) => {
  // немедленно выполнить обязательство
  resolve(123);
});

// впоследствии определить обработчик, который будет
// вызван немедленно, поскольку обязательство уже выполнено
p3.then(console.info);

// (info) 123
```

Определение нескольких обработчиков

Подобно тому, как один отложенный объект может быть разрешен или отклонен в нескольких местах приложения, так и у одного обязательства может быть несколько обработчиков, привязываемых к одному состоянию. Так, если к одному обязательству присоединено несколько обработчиков состояния "выполнено", то все они будут вызваны, когда это обязательство достигнет состояния "выполнено", как показано ниже. Это же справедливо и для обработчиков состояния "отклонено".

```
const p4 = new Promise((resolve, reject) => {
  // вызвать функцию resolve() через 1 секунду
  setTimeout(() => resolve(), 1000);
});

const cb = () => console.log('вызвано');

p4.then(cb);
p4.then(cb);
```

```
// Спустя 1 секунду:
// Сообщение "вызвано"
// Сообщение "вызвано"
```

Закрытые члены обязательства

Крайне важное усовершенствование по сравнению с Angular 1 состоит в том, что состояние обязательства вообще скрыто от процесса выполнения. Раньше состояние обязательства можно было извлечь из псевдозакрытого свойства `$$state`. А благодаря формальной реализации класса `Promise` в стандарте ES6 языка JavaScript состояние обязательства уже нельзя запросить в самом приложении. Но в то же время можно получить состояние обязательства из консоли. Например, в результате проверки обязательства в браузере Google Chrome получается нечто, похожее на следующее:

```
Promise {
  [[PromiseStatus]]: "fulfilled",
  [[PromiseValue]]: 123
}
```

Свойства **`PromiseStatus`** и **`PromiseValue`** относятся к закрытому типу **`Symbol`** и являются новыми конструкциями, появившимися в стандарте ES6 языка JavaScript. Тип **`Symbol`** можно рассматривать как однозначный ключ, удобный для установки свойств объектов, которые не должны быть доступны извне. Так, если символьная строка **`'PromiseStatus'`** служит в обязательстве ключом для доступа к свойству, ею легко воспользоваться за пределами объекта, даже если данное свойство должно оставаться закрытым. Но что касается закрытых символов в стандарте ES6 языка JavaScript, то при их создании они получают уникальное имя, и поэтому становятся недоступными в экземпляре объекта.

См. также

- Рецепт "Связывание обязательств и их обработчиков в цепочку", в котором подробно поясняется, как пользоваться эффективной конструкцией связывания в цепочку для последовательного выполнения асинхронных операций.
- Рецепт "Создание оболочек для обязательств с помощью методов `Promise.resolve()` и `Promise.reject()`", описывающий применение базовых служебных методов из класса `Promise`.

Связывание обязательств и их обработчиков в цепочку

Назначение обязательств состоит главным образом в том, чтобы дать разработчикам возможность последовательно анализировать и выполнять

независимые асинхронные операции. Этого можно добиться, применяя доступное в классе Promise средство связывания в цепочку.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/6828/**.

Реализация

Метод then(), определяющий обработчик состояний обязательства, возвращает другое обязательство, для которого могут быть установлены дополнительные обработчики в цепочке вызовов обработчиков:

```
var successHandler = () => { console.log('вызван'); };

var p = new Promise((resolve, reject) => { resolve(); })
  .then(successHandler)
  .then(successHandler)
  .then(successHandler);
// Сообщение "вызван"
// Сообщение "вызван"
// Сообщение "вызван"
```

Передача данных обработчикам по цепочке

Одни связанные в цепочку обработчики могут передавать данные другим обработчикам следующим образом:

```
var successHandler = (val) => {
  console.log(val);
  return val+1;
};

var p = new Promise((resolve, reject) => { resolve(0); })
  .then(successHandler)
  .then(successHandler)
  .then(successHandler);

// 0
// 1
// 2
```

Отклонение обработчиков по цепочке

Как правило, обычный возврат из обработчика состояний (но не исполнителя) будет по умолчанию служить сигналом, что все порожденные обязательства выполнены . Но если исполнитель или любой из последующих в цепочке обработчиков сгенерирует необрабатываемое исключение, то по

умолчанию обязательства будут отклонены. В приведенном ниже коде перехватывается это исключение.

```
const p = new Promise((resolve, reject) => {
  // Исполнитель немедленно сгенерирует исключение,
  // принуждающее к отклонению последующего обработчика
throw 123;
})
.then(
  // Обработчик состояния "выполнено" порожденного обязательства
  data => console.log('выполнено', data),
// Обработчик состояния "отклонено" порожденного обязательства
data => console.log('отклонено', data));

// Сообщения "отклонено", 123
```

Обратите внимание, что в данном случае исключение представляет собой примитивное числовое значение, передаваемые обработчику состояния "отклонено".

Принцип действия

Когда обязательство достигнет оконечного состояния, оно инициирует следующие по очереди порожденные обязательства. Этот простой, но весьма эффективный принцип позволяет создавать обширные и отказоустойчивые структуры, изящно объединяющие зависимые асинхронные действия.

Дополнение

Топология обязательств допускает некоторые интересные шаблоны применения.

Деревья обработчиков состояний обязательств

Обработчики состояний обязательств будут выполняться в том порядке, в каком определены обязательства. Если к единственному состоянию обязательства присоединено несколько обработчиков, то состояние будет обработано всеми его обработчиками, прежде чем удастся разрешить следующее по цепочке обязательство, как показано ниже.

```
const incr = val => {
  console.log(val);
  return ++val;
};

var outerResolve;
const firstPromise = new Promise((resolve, reject) => {
  outerResolve = resolve;
});
```

```
// определить обработчик состояний обязательства
// в переменной firstPromise
firstPromise.then(incr);

// присоединить еще один обработчик состояний обязательства
// в переменной firstPromise и накопить возвращаемое
// обязательство в переменной secondPromise
const secondPromise = firstPromise.then(incr);
// присоединить еще один обработчик состояний другого
// обязательства и накопить возвращаемое обязательство
// в переменной thirdPromise
const thirdPromise = secondPromise.then(incr);

// В данный момент вызов функции outerResolve() приведет
// к следующему:
// разрешение обязательства в переменной firstPromise;
// выполняются обработчики состояний данного обязательства;
// разрешение обязательства в переменной secondPromise;
// выполняются обработчики состояний данного обязательства;
// разрешение обязательства в переменной thirdPromise;
// обработчики его состояний еще не определены;

// порядок определения дополнительных обработчиков состояний
// обязательств не имеет особого значения; обязательства
// будут разрешены по мере определения их последующих состояний
secondPromise.then(incr);
firstPromise.then(incr);
thirdPromise.then(incr);

// В настоящий момент цепочка обязательств определена следующим образом:
// firstPromise -> secondPromise -> thirdPromise
// incr()          incr()           incr()
// incr()          incr()
// incr()
outerResolve(0);

// 0
// 0
// 0
// 1
// 1
// 2
```

Значение, возвращаемое обработчиком, определяет, выполнено или отклонено обязательство, и поэтому любые обработчики, связанные с обязательством, способны установить его состояние. Напомним, что состояние может быть установлено только один раз. Определение состояния родительского обязательства приведет к запуску обработчиков порожденных обязательств на выполнение.

Теперь должно быть ясно, каким образом деревья, образующие функциональные возможности обязательств, могут быть произведены из сочетания цепочек обязательств с цепочками обработчиков. Если пользоваться ими правильно, то можно получить весьма изящные решения трудных и неприятных задач последовательного выполнения асинхронных действий.

Метод `catch()`

Метод `catch()` является сокращенным вариантом метода `promise.then(null, errorCallback)`. Его применение может прояснить определение обязательств, хотя это не больше, чем синтаксическое удобство, как демонстрируется в следующем примере:

```
var outerReject;
const p = new Promise((resolve, reject) => {
  outerReject = reject;
})
.catch(() => console.log('отклонено!'));

outerReject();
// Сообщение "отклонено!"
```

Методы можно связать в цепочку **`p.then().catch()`**. Ошибка, генерируемая первоначальным обязательством, распространится через обязательство, созданное методом **`then()`**, приведет к его отклонению и достигнет обязательства, созданного методом **`catch()`**. При этом образуется еще один уровень косвенности обязательств, хотя для стороннего наблюдателя поведение останется прежним.

См. также

- Рецепт "Представление и реализация основных обязательств", подробно поясняющий, как и зачем пользоваться обязательствами.
- Рецепт "Создание оболочек для обязательств с помощью методов `Promise.resolve()` и `Promise.reject()`", в котором описано применение базовых служебных методов из класса `Promise`.

Создание оболочек для обязательств с помощью методов `Promise.resolve()` и `Promise.reject()`

Удобно иметь возможность создавать объекты обязательств, уже достигших оконечного состояния с определенным значением, а также нормализовать

объекты JavaScript, преобразовав их в обязательства. Оба эти действия позволяют выполнить методы `Promise.resolve()` и `Promise.reject()`.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/9315/**.

Реализация

Подобно всем остальным статическим методам из класса `Promise`, методы `Promise.resolve()` и `Promise.reject()` возвращают объект обязательства. В данном случае определение *исполнителя* отсутствует.

Если одному из этих методов передается любой другой аргумент, кроме обязательства, то возвращается обязательство в состоянии "выполнено" или "отклонено" соответственно. В следующем примере кода методам `Promise.resolve()` и `Promise.reject()`, кроме соответствующих обработчиков, передается также аргумент:

```
Promise.resolve('foo');
// Promise {[[PromiseStatus]]: "resolved",
            [[PromiseValue]]: "foo"}

Promise.reject('bar');
// Promise {[[PromiseStatus]]: "rejected",
            [[PromiseValue]]: "bar"}
// (ошибка) не обработана (в обязательстве) bar
```

Приведенный выше код по своему поведению равнозначен следующему коду:

```
new Promise((resolve, reject) => resolve('foo'));
// Promise {[[PromiseStatus]]: "resolved",
            [[PromiseValue]]: "foo"}

new Promise((resolve, reject) => reject ('bar'));
// Promise {[[PromiseStatus]]: "rejected",
            [[PromiseValue]]: "bar"}
// (ошибка) не обработана (в обязательстве) bar
```

Нормализация обязательств

Метод `Promise.resolve()` однозначно действует в тех случаях, когда объект обязательства передается ему в качестве аргумента. По существу, метод `Promise.resolve()` оперирует как пустая операция, возвращая первоначальный аргумент обязательства, вообще не модифицируя его. Он даже не попытается привести его к состоянию обязательства, заданного в качестве аргумента:

```
const a = Promise.resolve('baz');
console.log(a);
// Promise {status: 'resolved', value: 'baz'}

const b = Promise.resolve(a);
console.log(b);
// Promise {status: 'resolved', value: 'baz'}

console.log(a === b);
// истинно

const c = Promise.reject('qux');
// Сообщение об ошибке "qux"
console.log(c)
// Promise {status: 'rejected', value: 'qux'}

const d = Promise.resolve(c);
console.log(d);
// Promise {status: 'rejected', value: 'qux'}

console.log(c === d);
// истинно
```

Принцип действия

Если рассматривать обязательства в контексте того, что они, должны предоставить некое значение в будущем, то упомянутые выше методы просто улучшают любой латентный период, отделяя состояние ожидания от оконечного состояния. Такое разделение осуществляется очень просто.

- Метод `Promise.reject()` возвратит отклоненное обязательство независимо от того, какой аргумент ему передается. Даже если это объект обязательства, возвращаемым окажется значение объекта обязательства.
- Метод `Promise.resolve()` возвратит выполненное обязательство с заключенным в оболочку значением, если оно не является обязательством. А если это обязательство, то данный метод поведет себя как пустая операция.

Дополнение

Важно отметить, что метод `Promise.resolve()` ведет себя таким же образом, как и метод `$q.when()` в Angular 1. В частности, метод `$q.when()` был способен нормализовать объекты обязательств, но всегда возвращал вновь созданный объект обязательства, как показано ниже.

```
// Angular 1
const a = $q(() => {});
console.log(a);
```

```
// Promise {...}

const b = $q.when(a);
console.log(b);
// Promise {...}

console.log(a === b);
// ложно
```

См. также

- Рецепт "Представление и реализация основных обязательств", подробно поясняющий, как и зачем пользоваться обязательствами.
- Рецепт "Реализация барьеров для обязательств с помощью метода `Promise.all()`", в котором показано, как составлять композицию из обязательств.
- Рецепт "Отмена асинхронных действий с помощью метода `Promise.race()`", в котором поясняется весь процесс реализации полностью отказоустойчивой системы обязательств.

Реализация барьеров для обязательств с помощью метода `Promise.all()`

При разработке приложения может возникнуть потребность в применении обязательств в ситуации вроде "все или ничего". Это означает, что требуется сообща вычислить коллекцию обязательств, которая выполняется как единое обязательство в том и только в том случае, если будут выполнены все входящие в нее обязательства. А если любое из них отклоняется, то будет отклонено и совокупное обязательство.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **`http://ngcookbook.herokuapp.com/8496/`**.

Реализация

Методу `Promise.all()` передается итерируемая коллекция обязательств (например, массив объектов типа `Promise` или объект, имеющий ряд свойств обязательства), а он попытается выполнить все элементы данной коллекции как единое совокупное обязательство. В качестве параметра обработчику состояний совокупно выполняемого обязательства должен быть передан массив или объект, совпадающий с разрешаемыми значениями содержащихся в нем обязательств:

```
var outerResolveA, outerResolveB;
const promiseA = new Promise((resolve, reject) => {
  outerResolveA = resolve;
});
const promiseB = new Promise((resolve, reject) => {
  outerResolveB = resolve;
});

const multiPromiseAB = Promise.all([promiseA, promiseB])
  .then((values) => console.log(values));

outerResolveA(123);
outerResolveB(456);

// [123, 456]
```

Если отклоняется *любое* обязательство из коллекции, то отклоняется и совокупное обязательство. Параметр обработчика состояний совокупно разрешаемого обязательства примет значение, возвращаемое отклоненным обязательством, как демонстрируется в следующем примере кода:

```
var outerResolveC, outerRejectD;
const promiseC = new Promise((resolve, reject) => {
  outerResolveC = resolve;
});
const promiseD = new Promise((resolve, reject) => {
  outerRejectD = reject;
});

const multiPromiseCD = Promise.all([promiseC, promiseD])
  .then(
    values => console.log(values),
    rejectedValue => console.error(rejectedValue));

// выполнить совокупное обязательство, не запуская
// обработчик состояния "выполнено"
outerResolveC(123);
// отклонить совокупное обязательство, запустив обработчик
// состояния "отклонено"
outerRejectD(456);

// (ошибка) 456
```

Принцип действия

Как демонстрируется в данном рецепте, совокупное обязательство достигнет окончного состояния только в том случае, если будут выполнены все входящие в него обязательства, либо если хотя бы одно из этих обязательств будет отклонено. Пользоваться таким обязательством удобно в тех случаях,

когда обязательства в коллекции не должны анализировать друг друга, а единственной мерой удачного исхода для всей коллекции служит совместное завершение.

Если отклоняется входящее в коллекцию обязательство, то совокупное обязательство не будет ждать завершения остальных обязательств, хотя ничто не мешает им достичь своего оконечного состояния. Данные первого отклоняемого обязательства будет переданы обработчику состояний совокупного обязательства.

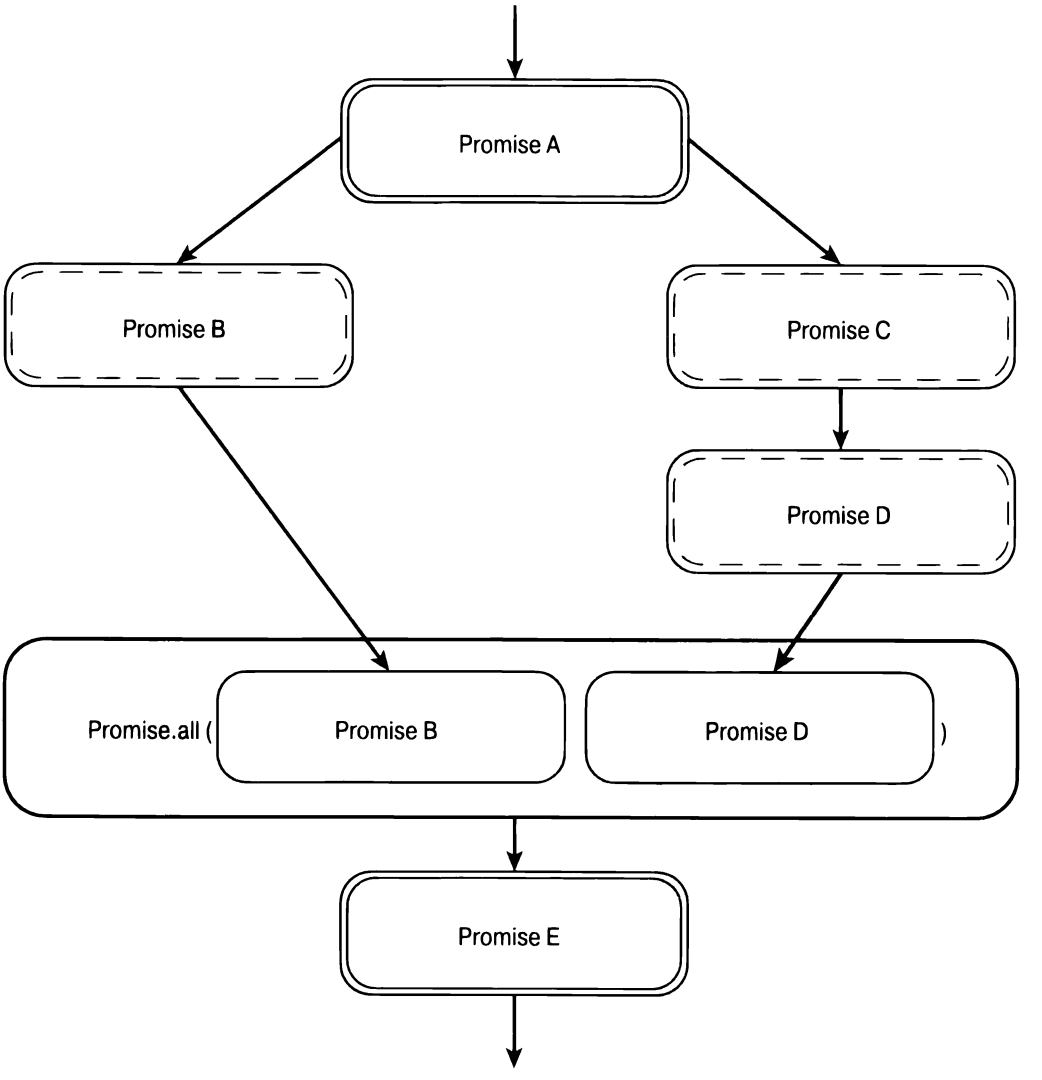
Дополнение

Во многих отношениях метод `Promise.all()` очень похож на барьер синхронизации процессов, действующих на уровне операционной системы. Такой барьер служит общей точкой сбора при выполнении потоковой инструкции, которого совокупность процессов достигает независимо и в разное время, и ни один процесс не продолжится дальше до тех пор, пока не будет достигнута данная точка. Аналогичным образом метод `Promise.all()` не продолжит свое выполнение, если только не будут выполнены все входящие в коллекцию обязательства (т.е. достигнут барьер), либо одно из обязательств не позволит вообще достичь этого состояния. И в этом случае управление возьмет на себя логика обработчика восстановления после отказа.

Метод `Promise.all()` дает возможность воссоединять обязательства, а следовательно, преобразовывать цепочки обязательств из приложения в *ориентированный ациклический граф* (ОАГ). В приведенном ниже примере демонстрируется граф сначала расходящейся, а затем сходящейся прогрессии обязательств. Такой уровень сложности необычен, хотя и доступен для применения, если потребуется в приложении.

См. также

- Рецепт "Создание оболочек для обязательств с помощью методов `Promise.resolve()` и `Promise.reject()`", демонстрирующий применение базовых служебных методов из класса `Promise`.
- Рецепт "Отмена асинхронных действий с помощью метода `Promise.race()`", в котором поясняется весь процесс реализации полностью отказоустойчивой системы обязательств.



Отмена асинхронных действий с помощью метода `Promise.race()`

В стандарте ES6 языка JavaScript внедрен метод `Promise.race()`, отсутствующий в спецификации службы `$q` из Angular 1. Подобно методу `Promise.all()`, этому статическому методу передается итерируемая коллекция объектов обязательств, выполнение или отклонение любого из которых становится результатом обработки всей коллекции обязательств в целом. Такое поведение может показаться необычным, тем не менее, оно приносит немалую пользу, когда требуется внедрить отменяющее поведение в систему.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/4362/**.

Подготовка

Допустим, разработка приложения начата с приведенного ниже простого обязательства, разрешаемого в значение через 3 секунды. При этом часть приложения желательно отделить от ожидания данного обязательства.

```
const delayedPromise = new Promise((resolve, reject) =>
                setTimeout(resolve.bind(null, 'foobar'), 3000))
               .then(val => console.log(val));
```

Реализация

В качестве простого решения можно было бы предоставить обработчик состояния "отклонено" данного обязательства и просто вызвать его откуда угодно, чтобы произвести отмену. Но лучше было бы прервать ожидание данного обязательства, а не разрушать его.

Конкретным тому примером может служить медленный, но крайне важный HTTP-запрос, который делается в веб-приложении. Чтобы не ожидать завершения операций в пользовательском интерфейсе, можно присоединить к запросу обработчики состояний, вызываемые после его завершения, и в которых будут обработаны полученные результаты.

Вместо этого можно выгодно воспользоваться методом `Promise.race()` и внедрить обязательство отмены наряду с первоначальным обязательством, как показано ниже.

```
// использовать следующий метод, чтобы зафиксировать
// функцию отмены

var cancel;

const cancelPromise = new Promise((resolve, reject) => {
  cancel = reject;
});
const delayedPromise = new Promise((resolve, reject) =>
  setTimeout(resolve.bind(null, 'foobar'), 3000));

// В методе Promise.race() создается новое обязательство
Promise.race([cancelPromise, delayedPromise])
  .then(
    val => console.log(val),
    () => console.error('отменено!'));
```

```
// Если вызвать метод cancel() до истечения 3 секунд,
// появится сообщение об ошибке "отменено!"

// А по истечении 3 секунд будет выведено
// сообщение "foobar" (нечто)
```

Если теперь обязательство `delayedPromise` будет выполнено первым, то обязательство, созданное методом `Promise.race()`, выведет на консоль переданное ему значение (в данном случае `"foobar"`). Но если вызвать метод `cancel()` прежде, чем это произойдет, тоже самое обязательство выведет сообщение об ошибке `"отменено!"`.

Принцип действия

Метод `Promise.race()` просто ожидает перехода в оконечное состояние любого из его внутренних обязательств. Он создает и возвращает новое обязательство, которое зависит от состояния этих внутренних обязательств. Когда же данный метод обнаружит, что любое из них перешло в оконечное состояние, в него перейдет и новое обязательство.

В данном примере исполнители обязательств **`cancelPromise`** и **`delayedPromise`** вызываются прежде метода **`Promise.race()`**. А поскольку одни обязательства интересуют лишь состояние других обязательств, то не имеет значения, что обязательства, передаваемые методу **`Promise.race()`**, должны быть уже запущены формально.

Следует, однако, иметь в виду, что метод `Promise.race()` не оказывает никакого воздействия на реализацию обязательства `delayedPromise`. И даже когда вызывается метод `cancel()`, обязательство `delayedPromise` будет по-прежнему выполняться, а обработчики его состояний — выполняться обычным образом, даже не подозревая, что объемлющий его метод `Promise.race()` уже отклонен. Можете убедиться в этом сами, добавив обработчик состояний "отклонено" к обязательству `delayedPromise`, вызвав метод `cancel()` и заметив, что этот обработчик все равно выполняется, как демонстрируется в следующем примере кода:

```
var cancel;
const cancelPromise = new Promise((resolve, reject) => {
  cancel = reject;
});
const delayedPromise = new Promise((resolve, reject) =>
  setTimeout(resolve.bind(null, 'foobar'), 3000))
  .then(() => console.log('тем не менее выполнено!'));

Promise.race([ cancelPromise, delayedPromise ])
  .then(
    val => console.log(val),
```

```
    () => console.error('отменено!'));

cancel();
// Сообщение "отменено!"

// А по истечении 3 секунд выводится
// сообщение "тем не менее выполнено! "
```

См. также

- Рецепт "Создание оболочек для обязательств с помощью методов `Promise.resolve()` и `Promise.reject()`", описывающий применение базовых служебных методов из класса `Promise`.
- Рецепт "Реализация барьеров для обязательств с помощью метода `Promise.all()`", в котором показано, как составлять композицию из обязательств.

Преобразование обязательств в наблюдаемые объекты

Наблюдаемые объекты и обязательства служат разным целям и пригодны в разных ситуациях, но в отдельной части приложения, как правило, приходится иметь дело с каким-то одним из этих средств. А это приводит к необходимости преобразования наблюдаемых объектов в обязательства, и наоборот. Правда, сделать это совсем не трудно благодаря наличию библиотеки RxJS.

Подробнее о поддержке наблюдаемых объектов в библиотеке RxJS см. страницу документации по адресу **http://reactivex.io/documentation/observable.html**.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/5244/**.

Реализация

У обязательств и наблюдаемых объектов можно найти немало общего. Имеются конкретные случаи удачного и ошибочного исхода, причем понятие успешного завершения соответствует только удачному исходу. Наблюдаемые объекты из библиотеки RxJS предоставляют метод `fromPromise()`, заключающий обязательство в оболочку как наблюдаемый объект:

```
import {Observable} from 'rxjs/Rx';

var outerResolve, outerReject;
```

```
const p1 = new Promise((resolve, reject) => {
  outerResolve = resolve;
  outerReject = reject;
});

var o1 = Observable.fromPromise(p1);
```

Итак, имея в своем распоряжении экземпляр наблюдаемого объекта типа `Observable`, теперь можно применить его метод `subscribe()` для обработки событий, соответствующих состояниям экземпляра обязательства типа `Promise`, как показано ниже.

```
import {Observable} from 'rxjs/Rx';

var outerResolve, outerReject;

const p1 = new Promise((resolve, reject) => {
  outerResolve = resolve;
  outerReject = reject;
});

var o1 = Observable.fromPromise(p1);

o1.subscribe(
  // Обработчик событий onNext()
  () => console.log('выполнено!'),
  // Обработчик событий onError()
  () => console.log('отклонено!'),
  // Обработчик событий onCompleted()
  () => console.log('завершено!'));

outerResolve();
// Сообщение "выполнено!"
// Сообщение "завершено!"
```

Принцип действия

Новый экземпляр наблюдаемого объекта типа `Observable` не заменяет обязательство, а только присоединяет себя к состояниям "выполнено" и "отклонено" обязательства. И когда это происходит, экземпляр наблюдаемого объекта генерирует событие и делает соответствующие обратные вызовы. Экземпляр типа `Observable` привязывается к состоянию обязательства, которому, впрочем, ничего об этом неизвестно, поскольку оно слепо предоставляет свои перехватчики состояний "выполнено" и "отклонено".

Учтите, что при выполнении обязательства будет вызван обработчик события **`onCompleted()`**. А при отклонении обязательства этот обработчик не вызывается.

Дополнение

Обязательства и наблюдаемые объекты можно считать взаимозаменяемыми, но не следует забывать, что они пригодны для разных ситуаций. В частности, наблюдаемые объекты пригодны в операциях потокового типа, где длина потока не определена. Безусловно, можно иметь наблюдаемый объект, генерирующий только одно событие, но этот наблюдаемый объект не распространит данное событие присоединенным далее приемникам, если только не настроить его надлежащим образом (например, в виде объекта типа `BehaviorObservable`).

Обязательства пригодны для маскирования асинхронного поведения. Они позволяют писать код и устанавливать обработчики состояний обязательства, как если бы обещанное состояние или значение было реализовано в момент выполнения. Это, конечно, не так, но возможность определять обработчик синхронно во время выполнения, чтобы принять в экземпляре типа `Promise` решение, когда удобнее всего выполнить этот обработчик, а также связывать обработчики в цепочку оказывается весьма ценной.

См. также

- Рецепт "Представление и реализация основных обязательств", подробно поясняющий, как и зачем пользоваться обязательствами.
- Рецепт "Преобразование наблюдаемого объекта HTTP-службы в обязательство типа `ZoneAwarePromise`", в котором дается ориентированное на Angular представление, как обязательства, наблюдаемые объекты и зоны интегрируются в приложение Angular.

Преобразование наблюдаемого объекта HTTP-службы в обязательство типа `ZoneAwarePromise`

Асинхронные наблюдаемые объекты из библиотеки RxJS являются объектами первого класса в Angular 2, и большая часть базового набора инструментальных средств настраивается на то, чтобы использовать их. Тем не менее взаимное преобразование наблюдаемых объектов и обязательств остается весьма ценным, поскольку у них сходные обязанности.

Подробнее о поддержке наблюдаемых объектов в библиотеке RxJS см. страницу документации по адресу **`http://reactivex.io/documentation/observable.html`**.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/0905/**.

Подготовка

Начните с приведенного ниже упрощенного варианта приложения. Допустим, что вызов HTTP-службы требуется преобразовать таким образом, чтобы воспользоваться обязательствами вместо наблюдаемых объектов.

```
[app/article.component.ts]

import {Component} from '@angular/core';
import {Http} from '@angular/http';

@Component({
  selector: 'article',
  template: `
    <p></p>
  `
})
export class ArticleComponent {
  constructor(private http:Http) {
    // Для целей демонстрации из этого простого запроса
    // исключены ошибки, возникающие при использовании ресурсов из
    // разных источников
    console.log(
      http.get('//run.plnkr.co/plunks/TBtcNDRelAOHDVpIuWw1'));
    }
  }

// Observable {...}
```

Реализация

Для целей данного рецепта совсем не обязательно разбираться в подробностях работы HTTP-службы и асинхронных наблюдаемых объектов из библиотеки RxJS. Достаточно знать, что любой метод, предоставляемый HTTP-службой, возвратит наблюдаемый объект. Правда, в реализации библиотеки RxJS может быть доступен метод `.toPromise()`, преобразующий наблюдаемые объекты в равнозначные им объекты типа `Promise`, как выделено ниже полужирным шрифтом.

```
[app/article.component.ts]

import {Component} from '@angular/core';
import {Http} from '@angular/http';
import 'rxjs/Rx';

@Component({
```

```
  selector: 'article',
  template: `
    <p></p>
  `
})
export class ArticleComponent {
  constructor(private http:Http) {
    // Для целей демонстрации из этого простого запроса
    // исключены ошибки использования ресурсов из
    // разных источников
    console.log(
    http.get('//run.plnkr.co/plunks/TBtcNDRelAOHDVpIuWw1')
        .toPromise());
  }
}
// ZoneAwarePromise {...}
```

Принцип действия

По умолчанию HTTP-служба возвращает наблюдаемый объект. Но в отсутствие импортируемого модуля `Rx` она сгенерирует ошибку, сообщая, что она не в состоянии обнаружить метод `.toPromise()`. Модуль `Rx` наделяет наблюдаемый объект возможностью преобразовываться в обязательство. Спецификация класса `Observable` со всеми операциями из RxJS намеренно не употребляется в Angular 2, чтобы указывать именно то, что требуется. Операции существуют в виде отдельных модулей, что приводит к сокращению объема полезной информации, отправляемой браузеру. Как только вызывается метод `.toPromise()`, создается экземпляр объекта типа `ZoneAwarePromise`.

На первый взгляд описанное выше преобразование кажется слишком сложным, но на самом деле это всего лишь заключение реализации класса **`Promise`** в оболочку библиотеки **`zone.js`**, чтобы в зоне Angular было известно о любых действиях, которые может вызвать обязательство. Для целей данного рецепта такую реализацию можно рассматривать как обычное обязательство.

См. также

- Рецепт "Представление и реализация основных обязательств", подробно поясняющий, как и зачем пользоваться обязательствами.
- Рецепт "Преобразование обязательств в наблюдаемые объекты", в котором представлен пример применения библиотеки RxJS для взаимного преобразования этих двух эффективных типов объектов.

Глава 5

Наблюдаемые объекты из библиотеки ReactiveX

В этой главе даются следующие рецепты.

- Простое применение наблюдаемых объектов в HTTP-запросах
- Реализация модели "издатель–подписчик" средствами класса `Subject`
- Создание наблюдаемой службы аутентификации средствами класса `BehaviorSubject`
- Создание обобщенной службы "издатель–подписчик" для замены служб `$broadcast`, `$emit` и `$on`
- Отслеживание изменений в декораторе `@ViewChildren` средствами классов `QueryList` и `Observable`
- Создание полноценного компонента `AutoComplete` с помощью наблюдаемых объектов

Введение

Чтобы понять суть наблюдаемых объектов в Angular 2, очень важно уяснить задачу, которую требуется решить. В процессе разработки программного обеспечения нередко возникает ситуация, когда какой-нибудь компонент должен известить все остальные компоненты о чем-то произошедшем. Назовем это "событием", в отличие от события в браузере. В таком случае было бы желательно подключиться к компоненту, присоединив к нему соответствующее поведение при наступлении события. Было бы также желательно отключиться от этого компонента, если извещаемые им события больше не интересуют.

В этой главе рассматриваются дополнительные сложности и особенности применения наблюдаемых объектов. Но такое представление о событиях

особо подчеркивает полезность для разработчиков основополагающего шаблона.

Шаблон "Наблюдатель"

Шаблон "Наблюдатель" (Observer) не является библиотекой или каркасом. Это всего лишь проектный шаблон, используемый при разработке программного обеспечения, на котором основываются наблюдаемые объекты из библиотеки ReactiveX. Этот шаблон реализован во многих языках программирования и библиотеках. К их числу относится библиотека ReactiveX, формально внедренная в каркасе Angular 2.

В шаблоне "Наблюдатель" описывается взаимосвязь между *субъектом*, иначе называемым "логическим объектом", и его *наблюдателями*. Субъекту известны любые наблюдатели, которые следят за ним. Когда распространяется событие, субъект в состоянии передать его каждому наблюдателю через методы, предоставляемые в тот момент, когда наблюдатель начинает подписку на него.

Библиотеки ReactiveX и RxJS

Библиотека ReactiveX реализована на самых разных языках, включая Python, Java и Ruby. А реализация библиотеки ReactiveX в языке JavaScript, называемая RxJS, представляет собой зависимость, употребляемую в Angular 2 для внедрения наблюдаемых объектов в естественное поведение этого каркаса. Подобно обязательствам, самостоятельный экземпляр типа `Observable` может быть создан непосредственно, но по умолчанию интерфейс наблюдаемых объектов типа `Observable` также используются во многих методах и службах в Angular 2.

Наблюдаемые объекты в Angular 2

Наблюдаемые объекты внедряются в Angular 2 самыми разными способами. Если они для вас внове, то пользоваться ими поначалу будет неудобно. Тем не менее следует признать, что наблюдаемые объекты предоставляют более совершенный шаблон для разработки программного обеспечения.

Наряду с крупным модулем `rxjs/Rx` из библиотеки RxJS в распоряжение разработчиков предоставляется урезанный модуль `rxjs/Observable`. Этот минимальный модуль позволяет импортировать по мере надобности отдельные части несущественного поведения, чтобы сократить разрастание модуля. Например, применяя операции (`operators`) и прочие средства библиотеки ReactiveX наряду с упрощенным модулем `Observable`, приходится явным образом внедрять эти средства и модули, чтобы расширить имеющийся интерфейс наблюдаемых объектов типа `Observable`.

Наблюдаемые объекты и обязательства

Как наблюдаемые объекты, так и обязательства предоставляют решения для построения асинхронных конструкций, но наблюдаемые объекты оказываются в этом отношении более надежными, расширяемыми и полезными. И хотя обязательства доступны в спецификации стандарта ES6 языка JavaScript по умолчанию, тем не менее, очень быстро становится ясно, что они оказываются малопригодными при попытке применить их за пределами основного поведения разрабатываемого приложения.

В библиотеке ReactiveX предоставляется эффективный набор инструментальных средств, с которым не могут сравниться обязательства. Наблюдаемые объекты можно составлять в композиции, что дает возможность преобразовывать и объединять их с новыми наблюдаемыми объектами. Кроме того, они инкапсулируют понятие непрерывного потока событий, т.е. парадигму, часто встречающуюся в программировании клиентских приложений, тогда как обязательства не вполне вписываются в нее.

Простое применение наблюдаемых объектов в HTTP-запросах

Шаблон "Наблюдатель" по умолчанию применяется в модуле `Http` из Angular 2 для заключения в оболочку объекта типа `XMLHttpRequest`. Разработчикам, знакомым с этим шаблоном, следует знать, что его легко приспособить к асинхронному характеру запросов удаленных объектов. А начинающим осваивать шаблон "Наблюдатель" следует изучить особенности применения наблюдаемых объектов из модуля `Http`, чтобы лучше понять эту новую парадигму.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **`http://ngcookbook.herokuapp.com/4121/`**.

Подготовка

Ради целей данного рецепта в рассматриваемом здесь примере будет использован статический файл формата JSON. Но точно так же можно было бы организовать отправку запросов в динамическом режиме.

Итак, начните с создания следующей заготовки компонента, включая все модули, необходимые для составления HTTP-запросов:

```
[app/article.component.ts]

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

```
@Component({
  selector: 'article',
  template: `
    <h1>{{title}}</h1>
    <p>{{author}}</p>
  `
})
export class ArticleComponent {
  title:string;
  body:string;
  constructor (private http: Http) {
  }
}
```

Допустим, что для целей данного примера в каталоге `static` находится следующий статический файл `article.json` формата JSON:

```
[article.json]

{
  "title": "Orthopedic Doctors Ask City for More Sidewalk Cracks",
  "author": "Jake Hsu"
}
```

Реализация

Служба `Http` уже внедрена, и поэтому можете приступить к определению запроса по методу `GET`, как выделено ниже полужирным шрифтом. При этом создается экземпляр типа `Observable`, но это не избавляет от необходимости вводить инструкции по обработке исходных символьных строк, получаемых в ответ.

```
[app/article.component.ts]

import {Component} from '@angular/core';
import {Http} from '@angular/http';

@Component({
  selector: 'article',
  template: `
    <h1>{{title}}</h1>
    <p>{{author}}</p>
  `
})
export class ArticleComponent {
  title:string;
  body:string;
  constructor (private http_: Http) {
    http_.get('static/article.json');
  }
}
```

В данный момент вы можете заметить, что это на самом деле не приводит к инициализации запроса по методу **GET** в браузере. Подробнее об этом речь пойдет далее, в разделе "Дополнение" данного рецепта.

Известно, что по составленному выше запросу возвратятся данные в формате JSON, и поэтому можно употребить метод `json()`, предоставляемый в классе `Response`. И хотя это можно сделать в теле метода `map()`, последний, тем не менее, не предоставляется в классе `Observable` по умолчанию, а следовательно, его придется импортировать из модуля `rxjs`, как показано ниже.

```
[app/article.component.ts]

import {Component} from '@angular/core';
import {Http} from '@angular/http';
import 'rxjs/add/operator/map';

@Component({
  selector: 'article',
  template: `
    <h1>{{title}}</h1>
    <p>{{author}}</p>
  `
})
export class ArticleComponent {
  title:string;
  author:string;
  constructor (private http_: Http) {
    http_.get('static/article.json')
         .map(response => response.json());
  }
}
```

Все вроде бы хорошо, но не все еще сделано. В приведенном выше фрагменте кода создается экземпляр типа `Observable`, но на него еще нужно подписаться, чтобы можно было обрабатывать любые отправляемые им данные. И это можно сделать с помощью метода `subscribe()`, позволяющего присоединить методы наблюдателя для обратного вызова и обработки ошибок, как демонстрируется в приведенном ниже фрагменте кода. Таким образом, из запроса по методу `GET` возвратится файл типа JSON, и ответные данные будут синтаксически проанализированы и вставлены компонентом в модель DOM.

```
[app/article.component.ts]

import {Component} from '@angular/core';
import {Http} from '@angular/http';
import 'rxjs/add/operator/map';

@Component({
```

```
  selector: 'article',
  template: `
    <h1>{{title}}</h1>
    <p>{{author}}</p>
  `
})
export class ArticleComponent {
  title:string;
  author:string;
  constructor (private http_: Http) {
    http_.get('static/article.json')
         .map(response => response.json())
         .subscribe(
           article => {
             this.title = article.title;
             this.author = article.author;
           },
           error => console.error(error));
  }
}
```

Принцип действия

В предыдущем разделе был сделан довольно общий обзор происходящего при обработке HTTP-запроса. Но было бы полезно более тщательно проанализировать рассматриваемый здесь рецепт, чтобы стало понятнее каждое предпринимаемое в нем действие.

Обобщенный тип `Observable<Response>`

В классе службы `Http` предоставляются методы `get()`, `post()`, `put()` и прочие, по существу, обозначающие, как и следовало ожидать, терминологию сетевого протокола HTTP. Каждый из этих методов возвратит приведенный ниже обобщенный тип `Observable<Response>`, предоставляющий экземпляр типа `Response` при возврате из запроса.

```
console.log(http_.get('static/article.json'));
// Observable { ... }
```

Вполне очевидно, что наблюдаемые объекты должны отслеживаться *наблюдателем*. В частности, наблюдатель будет ожидать объекты, отправляемые наблюдаемым объектом, и в данном примере они принимают форму экземпляра типа **Response**.

Метод `map()` из библиотеки RxJS

Экземпляр типа `Response` предоставляет метод `json()`, преобразующий возвращаемую сериализированную символьную строку с полезной информацией в соответствующее ее объектное представление в оперативной

памяти. Было бы желательно передать обычный объект обработчику событий в наблюдателе, и для этой цели идеально подходит клиновидный метод, по-прежнему возвращающий наблюдаемый объект:

```
console.log(http_.get('static/article.json')
       .map(response => response.json()));
// Observable {source: Observable, operator: MapOperator, ...}
```

Напомним, что в канонической форме наблюдаемые объекты являются потоком событий. В данном случае известно, что в качестве единственного вообще события ожидается HTTP-ответ. Тем не менее все операции, обычно выполняемые над потоком событий, могут быть в такой же мере выполнены и над наблюдаемым объектом этого единственного события.

Аналогичным образом метод `Array.map()` может быть использован для преобразования каждого экземпляра в массиве. А метод `Observable.map()` позволяет преобразовать каждое событие, исходящее из наблюдаемого объекта. В частности, он создает еще один наблюдаемый объект, распространяющий модифицированное событие, поступающее от первоначального наблюдаемого объекта.

Подписка

Экземпляры типа `Observable` предоставляют метод `subscribe()`, которому в качестве аргументов передаются обработчики событий `onNext()`, `onError()` и `onCompleted()`. Эти обработчики соответствуют событиям жизненного цикла объекта типа `Observable`, когда тот начинает отправлять объекты типа `Response`. В качестве параметра методу `onNext()` передается все то, что исходит от наблюдаемого объекта. В данном случае в качестве отправляемых данных служит значение, возвращаемое методом `map()`. Следовательно, это будет синтаксически проанализированный объект, возвращаемый после вызова метода `json()` для экземпляра типа `Response`. Все эти методы необязательны, но в данном примере методы `onNext()` и `onError()` полезны.

Когда все упомянутые выше методы обработки событий передаются методу **`subscribe()`** в качестве аргументов, они совместно образуют то, что называется *наблюдателем*.

```
http_.get('static/article.json')
     .map(respose => respose.json())
     .subscribe(
        article => {
          this.title = article.title;
          this.body = article.body;
        },
        error => console.error(error));
```

Принимая все сказанное выше во внимание, следует заметить, что браузер извлечет данные в формате JSON и синтаксически проанализирует их, а подписчик передаст свои данные соответствующим членам компонента.

Дополнение

Постепенно прорабатывая данный рецепт и наблюдая, как составляемые вами сетевые запросы обрабатываются в браузере, вы, вероятно, заметите, что конкретный запрос по методу `GET` не инициируется до тех пор, пока не будет вызван метод `subscribe()`. Дело в том, что используемый вами тип `Observable` оказывается "холодным".

"Горячие" и "холодные" наблюдаемые объекты

Определение "холодный" обозначает, что наблюдаемый объект не начнет распространять событие до тех пор, пока наблюдатель не подпишется на него. В отличие от "холодного" наблюдаемого объекта, "горячий" наблюдаемый объект распространяет событие, даже если на него не подписаны наблюдатели. А поскольку это означает, что события, происходящие прежде присоединения наблюдателя, теряются, то наблюдаемые объекты требуется обозначать как "холодные" в HTTP-запросах.

Метод `onNext()` называется "эмиссионным"|, поскольку имеются связанные с ним распространяемые данные. А методы `onCompleted()` и `onError()` называются "уведомительными", потому что они представляют нечто значимое, хотя с ними не связано событие, которое будет считаться частью потока.

См. также

- Рецепт "Реализация модели "издатель–подписчик" средствами класса `Subject`", в котором демонстрируется, как настроить ввод-вывод для наблюдаемых объектов из библиотеки RxJS.
- Рецепт "Создание полноценного компонента `AutoComplete` с помощью наблюдаемых объектов", в котором дается обширный обзор некоторых служебных методов, предоставляемых в библиотеке RxJS.

Реализация модели "издатель–подписчик" средствами класса `Subject`

Зачастую в Angular 2 свободно предоставляется интерфейс наблюдаемых объектов типа `Observable` для присоединения к наблюдаемым объектам, но очень важно понимать, каким образом эти объекты создаются, настраиваются и применяются. В частности, полезно знать, как применять наблюдаемые объекты в реальных ситуациях, возникающих на стороне клиента.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/4839/**.

Подготовка

Допустим, разработка приложения начата с приведенной ниже заготовки. Цель состоит в том, чтобы регистрировать в повторяющемся поле все события от щелчков кнопками мыши на экранной кнопке.

```
[app/click-observer.component.ts]

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

@Component({
  selector: 'click-observer',
  template: `
    <button>
      Emit event!
    </button>
    <p *ngFor="let click of clicks; let i = index">
      {{i}}: {{click}}
    </p>
  `
})
export class ClickObserverComponent {
  clicks:Array<Event> = [];
}
```

Реализация

Чтобы решить поставленную выше задачу, можно привязать компонентный метод к событию `click` в шаблоне, но это не позволит извлечь реальную выгоду из наблюдаемых объектов. В данном случае требуется сделать наблюдаемый объект доступным в компоненте `ClickObserverComponent`. Благодаря этому подписаться на события `click` и обработать их по-своему можно будет в любой другой части приложения.

Вместо этого было бы желательно направить события `click` от экранной кнопки к наблюдаемому объекту. Но сделать это с помощью обычного экземпляра типа `Observable` невозможно, поскольку он действует лишь как часть, отвечающая за *подписку* в модели "издатель–подписчик". Чтобы реализовать другую часть этой модели, отвечающую за *издание*, необходимо воспользоваться экземпляром типа `Subject` следующим образом:

```
[app/click-observer.component.ts]

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

```
import {Subject} from 'rxjs/Subject';

@Component({
  selector: 'click-observer',
  template: `
    <button>
      Emit event!
    </button>
    <p *ngFor="let click of clicks; let i = index">
      {{i}}: {{click}}
    </p>
  `
})
export class ClickObserverComponent {
  clickEmitter:Subject<Event> = new Subject();
  clicks:Array<Event> = [];
}
```

Субъекты типа `Subject` из библиотеки ReactiveX действуют как наблюдаемые объекты и как наблюдатель. Следовательно, они предоставляют метод `subscribe()`, определяющий поведение подписчика, а также метод `next()`, определяющий поведение издателя.

В данном примере пригоден метод **`next()`**, поскольку требуется явным образом указать, когда должно произойти распространение и что должно быть распространено. Имеется немало способов получить экземпляры типа **`Observable`**, чтобы неявно сформировать распространение, например, в виде метода **`Observable.range()`**. В подобных случаях наблюдаемому объекту известно поведение его входных данных, а следовательно, ему не требуется указывать момент распространения и что должно быть распространено.

В данном случае событие можно передать непосредственно методу `next()` в определении обработчика событий `click` в шаблоне. Следовательно, остается лишь заполнить массив, направив в него распространяемые данные, как показано ниже.

```
[app/click-observer.component.ts]

import {Component} from '@angular/core';
import {Subject} from 'rxjs/Subject';

@Component({
  selector: 'click-observer',
  template: `
    <button (click)="clickEmitter.next($event)">
      Emit event!
    </button>
    <p *ngFor="let click of clicks; let i = index">
```

```
      {{i}}: {{click}}
    </p>
  `
})
export class ClickObserverComponent {
  clickEmitter:Subject<Event> = new Subject();
  clicks:Array<Event> = [];

  constructor() {
    this.clickEmitter
        .subscribe(clickEvent =>
                   this.clicks.push(clickEvent));
  }
}
```

Вот, собственно, и все! Таким образом, события `click` будут отображаться в браузере при каждом очередном щелчке на экранной кнопке.

Принцип действия

Наблюдаемые объекты в библиотеке ReactiveX отличаются от наблюдателей, но их поведение взаимно совместимо настолько, что их объединение (`Subject`) может действовать как один из них. В данном примере субъект типа `Subject` служит в качестве интерфейса для подачи объектов-событий типа `Event` в качестве издающей составляющей модели "издатель–подписчик", а также может обрабатывать их результат в качестве подписной части этой модели.

Дополнение

Такой способ построения в данном рецепте может показаться необычным. Компонент предоставляет экземпляр типа `Subject` в качестве точки, где приложение должно присоединить наблюдатели.

Тем не менее требуется, чтобы дополнительные события не добавлялись из других частей приложения, что по-прежнему возможно, если воспользоваться в них методом `next()`. Кроме того, ссылка на экземпляр типа `Subject` делается непосредственно в шаблоне, и поэтому его доступность там может оказаться излишней. Поэтому было бы желательно и, конечно, правильно сделать доступным только наблюдаемый объект как составляющую субъекта.

Для этого необходимо импортировать модуль `Observable` и употребить компонентный метод `asObservable()`, предоставляемый экземпляром типа `Subject`. Этот метод создаст новый экземпляр типа `Observable`, который, по существу, направит наблюдаемые распространяемые данные по каналу от субъекта к новому наблюдаемому объекту, который станет доступным в качестве открытого члена компонента, как показано ниже. Теперь каждый

компонент, ссылающийся на свойство clickEmitter, совсем необязательно должен обращаться к субъекту как к первоисточнику, несмотря на то, что только данный компонент ссылается на свойство clickEmitter.

[app/article.component.ts]

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

@Component({
  selector: 'click-observer',
  template: `
    <button (click)="publish($event)">
      Emit event!
    </button>
    <p *ngFor="let click of clicks; let i = index">
      {{i}}: {{click}}
    </p>
  `
})
export class ClickObserverComponent {
  clickEmitter: Observable<Event>;
  private clickSubject_: Subject<Event> = new Subject();
  clicks:Array<Event> = [];

  constructor() {
    this.clickEmitter = this.clickSubject_.asObservable();
    this.clickEmitter.subscribe(clickEvent =>
    this.clicks.push(clickEvent));
  }

  publish(e:Event):void {
    this.clickSubject_.next(e);
  }
}
```

Собственная реализация в библиотеке RxJS

Все представленное выше послужило отличным примером применения шаблона "Наблюдатель", но он настолько распространен, что в библиотеке RxJS уже предоставляется встроенный способ его реализации. Так, в классе Observable доступен метод fromEvent(), которому в качестве аргументов передается элемент, который, как ожидается, должен генерировать события, а также тип принимаемых событий.

Для работы вам потребуется ссылка на конкретный элемент, который в настоящий момент отсутствует. Для текущей реализации декоратор @ViewChild в Angular 2 предоставит весьма изящную ссылку на кнопку, которая затем

передается методу fromEvent(), как только шаблон будет отображен и как показано ниже. Таким образом, компонент должен вести себя, как и прежде.

```
[app/click-observer.component.ts]

import {Component, ViewChild, ngAfterViewInit}
        from '@angular/core';
import {Observable} from 'rxjs/Observable';
import 'rxjs/add/observable/fromEvent';

@Component({
  selector: 'click-observer',
  template: `
    <button #btn>
      Emit event!
    </button>
    <p *ngFor="let click of clicks; let i = index">
      {{i}}: {{click}}
    </p>
  `
})
export class ClickObserverComponent implements AfterViewInit {
  @ViewChild('btn') btn;
  clickEmitter:Observable<Event>;
  clicks:Array<Event> = [];

  ngAfterViewInit() {
    this.clickEmitter = Observable.fromEvent(
        this.btn.nativeElement, 'click');
    this.clickEmitter.subscribe(clickEvent =>
        this.clicks.push(clickEvent));
  }
}
```

См. также

- Рецепт "Простое применение наблюдаемых объектов в HTTP-запросах", демонстрирующий основы применения интерфейса наблюдаемых объектов.
- Рецепт "Создание наблюдаемой службы аутентификации средствами класса BehaviorSubject", в котором поясняется, как реактивно управлять состоянием в приложении.
- Рецепт "Создание полноценного компонента AutoComplete с помощью наблюдаемых объектов" с обширным обзором некоторых служебных методов, предоставляемых в библиотеке RxJS.

Создание наблюдаемой службы аутентификации средствами класса BehaviorSubject

К числу самых очевидных и полезных примеров применения шаблона "Наблюдатель" относится однонаправленная передача информации из одного логического объекта приложения во внешнее поле приемников. Эти приемники должны быть способны свободно присоединяться и отсоединяться от распространяющего информацию логического объекта. Характерным тому первоначальным примером служит компонент входа и выхода из системы.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/6957/**.

Подготовка

Допустим, разработка приложения начата со следующей заготовки:

```
[app/login.component.ts]

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

@Component({
  selector: 'login',
  template: `
    <button *ngIf="!loggedIn"
            (click)="loggedIn=true">
      Login
    </button>
    <button *ngIf="loggedIn"
            (click)="loggedIn=false">
      Logout
    </button>
  `
})
export class LoginComponent {
  loggedIn:boolean = false;
}
```

В настоящем виде данный компонент позволяет выбирать одну из экранных кнопок входа и выхода из системы, но в нем отсутствует представление об общем состоянии приложения, и поэтому в других компонентах нельзя употреблять состояние входа в систему, которое должно отслеживаться в данном компоненте. Цель состоит в том, чтобы внедрить это состояние в общую службу, приводимую в действие по шаблону "Наблюдатель".

Реализация

Начните с создания пустой службы и ее внедрения в данный компонент, как показано ниже. Обратите внимание на то, что в данном случае применяется конструкция `const enum` языка TypeScript в виде перечисления для хранения состояния аутентификации пользователя.

```
[app/authentication.service.ts]

import {Injectable} from '@angular/core';

@Injectable()
export class AuthService {
  private authState_: AuthState;
}

export const enum AuthState {
  LoggedIn,
  LoggedOut
}
```

Если вы только начинаете осваивать новый стандарт ES6 языка JavaScript, а также язык TypeScript, приведенные выше ключевые слова могут показаться вам не совсем обычными. В частности, ключевое слово **const**, описываемое в стандарте ES6, обозначает, что данное значение доступно только для чтения, как только оно будет объявлено. В реализации языка JavaScript по стандарту ES6 это приведет к генерированию ошибки (как правило, **SyntaxError**) во время выполнения приведенного выше кода. Но при компиляции исходного кода на языке TypeScript ключевое слово **const** будет перехвачено во время компиляции.

Ключевое слово **enum** предоставляется в языке TypeScript. Оно непохоже на обычный объектный литерал. Следует, однако, иметь в виду, что у членов перечисления, определяемого ключевым словом **enum**, отсутствуют конкретные значения.

Обращение к членам перечисления в рассматриваемом здесь приложении будет осуществляться с помощью вызываемых методов **AuthState.LoggedIn()** и **AuthState.LoggedOut()**. Если вы обратитесь к ним в скомпилированном коде JavaScript, сгенерированном в TypeScript, то обнаружите, что им фактически присвоены целочисленные значения. Но при разработке крупных приложений это дает возможность создать централизованное хранилище данных перечислимого типа **AuthState**, не особенно беспокоясь об их фактических значениях.

Внедрение службы аутентификации

В текущей заготовке данной службы требуется получить экземпляр типа `Subject`, который будет распространять данные перечислимого типа

AuthState о состоянии аутентификации, но в настоящий момент взаимодействие с ним отсутствует, и поэтому его необходимо каким-то образом организовать. Прежде всего внедрите данную службу в свой компонент следующим образом:

[app/app.module.ts]

```
import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {LoginComponent} from './login.component';
import {AuthService} from './authentication.service';

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

Все это замечательно и прекрасно, но данная служба по-прежнему непригодна для применения в настоящем виде.

Путь для импорта класса **AuthService** может меняться в зависимости от местонахождения его исходного файла в дереве каталогов.

Ввод объекта типа BehaviorSubject в службу аутентификации

Назначение данной службы — поддерживать глобальное состояние приложения. Она должна быть доступна в остальной части приложения, чтобы эти части могли сделать ей запрос на извещение об изменении состояния, а также о текущем состоянии. И для этой цели идеально подходит класс BehaviorSubject.

У класса **Subject** из библиотеки RxJS имеется несколько подклассов, к числу которых относится класс **BehaviorSubject**. Он наследует, по существу, все особенности класса **Subject**, но главное его отличие заключается в том, что он передает свое текущее состояние любому наблюдателю, начинающему принимать его как совершенно новое событие. И это очень удобно в тех случаях, когда требуется отслеживать состояние.

Добавьте закрытый объект типа BehaviorSubject, инициализированный состоянием LoggedOut, а также открытый объект типа Observable в класс AuthService, как показано ниже.

[app/authentication.service.ts]

```
import {Injectable} from '@angular/core';
import {BehaviorSubject} from 'rxjs/BehaviorSubject';
import {Observable} from 'rxjs/Observable';

@Injectable()
export class AuthService {
  private authManager_:BehaviorSubject<AuthState>
          = new BehaviorSubject(AuthState.LoggedOut);
  private authState_:AuthState;
  authChange:Observable<AuthState>;

  constructor() {
    this.authChange = this.authManager_.asObservable();
  }
}

export const enum AuthState {
  LoggedIn,
  LoggedOut
}
```

Добавление методов из интерфейса API в службу аутентификации

Напомним, что экземпляр типа BehaviorSubject не должен быть доступен внешним действующим лицам. Вместо этого желательно предоставить только его компонент типа Observable, на который можно открыто подписаться. Желательно также дать внешним действующим лицам возможность установить состояние аутентификации, но только косвенно. И этого можно достигнуть с помощью методов, выделенных ниже полужирным шрифтом.

[app/authentication.service.ts]

```
import {Injectable} from '@angular/core';
import {BehaviorSubject} from 'rxjs/BehaviorSubject';
import {Observable} from 'rxjs/Observable';

@Injectable()
export class AuthService {
  private authManager_:BehaviorSubject<AuthState>
          = new BehaviorSubject(AuthState.LoggedOut);
  private authState_:AuthState;
```

```
  authChange:Observable<AuthState>;
  constructor() {
    this.authChange = this.authManager_.asObservable();
  }
  login():void {
    this.setAuthState_(AuthState.LoggedIn);
  }
  logout():void {
    this.setAuthState_(AuthState.LoggedOut);
  }
  emitAuthState():void {
    this.authManager_.next(this.authState_);
  }
  private setAuthState_(newAuthState:AuthState):void {
    this.authState_ = newAuthState;
    this.emitAuthState();
  }

}
export const enum AuthState {
  LoggedIn,
  LoggedOut
}
```

Превосходно! Таким образом, внешние действующие лица смогут подписываться на наблюдаемый объект `authChange` и косвенно контролировать состояние аутентификации через методы `login()` и `logout()`.

Следует иметь в виду, что наблюдаемому объекту, составляющему субъект типа **BehaviorSubject**, присвоено имя **authChange**. Присвоить имена различным составляющим элементов в шаблоне "Наблюдатель" нелегко. Имя **authChange** было выбрано с целью обозначить назначение события, исходящего из наблюдаемого объекта. Оно буквально дает ответ на вопрос: "Какое событие наблюдается?" Следовательно, имеет вполне семантический смысл подписывать компонент на события **authChange**, когда изменяется состояние аутентификации.

Подключение служебных методов к компоненту

В классе `LoginComponent` пока еще не используется рассматриваемая здесь служба, поэтому добавьте в него следующие новые методы:

```
[app/login.component.ts]

import {Component} from '@angular/core';
import {AuthService, AuthState} from './authentication.service';

@Component({
```

```
  selector: 'login',
  template: `
    <button *ngIf="!loggedIn"
            (click)="login()">
      Login
    </button>
    <button *ngIf="loggedIn"
            (click)="logout()">
      Logout
    </button>
  `
})
export class LoginComponent {
  loggedIn:boolean;

  constructor(private authService_:AuthService) {
    authService_.authChange.subscribe(
      newAuthState => this.loggedIn
                    = (newAuthState === AuthState.LoggedIn));
  }

  login():void {
    this.authService_.login();
  }

  logout():void {
    this.authService_.logout();
  }
}
```

А теперь, когда все находится на своих местах, экранные кнопки входа и выхода из системы должны функционировать должным образом. Это означает, что вы правильно внедрили наблюдаемый объект в свой компонент.

Данный рецепт служит характерным примером тех соглашений, которых необходимо придерживаться, используя открытые и закрытые члены. В данном случае внедренная служба объявляется как закрытый член и заключается в оболочку с помощью открытых компонентных методов. А все, что вызывается в остальной части приложения или применяется в шаблоне, должно быть открытым членом.

Принцип действия

Главное в рассматриваемой здесь реализации заключается в том, что каждый компонент, прослушивающий наблюдаемый объект, выполняет обработку идемпотентных распространяемых событий. Всякий раз, когда новый компонент связывается с наблюдаемым объектом, он предписывает службе распространить данные о текущем состоянии, используя метод

emitAuthState(). Необходимо, чтобы поведение всех компонентов не изменялось в случае, если они обнаружат, что данные об одном и том же состоянии распространяются несколько раз подряд. Они должны изменить свое поведение лишь в том случае, если будет обнаружено изменение в состоянии. Обратите внимание на то, что в данном рецепте состояние аутентификации полностью внедрено в службу аутентификации, и в то же время предоставлен для применения реактивный интерфейс API, на основании которого написано все рассматриваемое здесь приложение.

Дополнение

Двумя крайне важными составляющими подключения к службам, подобным рассматриваемой в данном рецепте, являются процессы установки и удаления. Дотошный читатель заметит, что даже если уничтожить экземпляр типа LoginComponent, то подписка на наблюдаемый объект все равно останется. А это, разумеется, крайне нежелательно!

Правда, метод subscribe() из класса Observable возвращает экземпляр типа Subscription, предоставляющий метод unsubscribe(). Следовательно, этот экземпляр можно зафиксировать после вызова метода subscribe(), а затем вызвать его метод unsubscribe(), когда удаляется компонент.

Аналогично удалению приемника в Angular 1, необходимо вызвать метод unsubscribe(), когда уничтожается экземпляр компонента. Правда, в жизненном цикле Angular 2 предоставляется аналогичный метод ngOnDestroy(), где можно вызвать метод unsubscribe(), как показано ниже. В итоге вы сможете обезопасить свое приложение от утечек памяти, если любой экземпляр данного компонента уничтожается в жизненном цикле вашего приложения.

[app/login.component.ts]

```
import {Component, ngOnDestroy} from '@angular/core';
import {AuthService, AuthState} from './authentication.service';
import {Subscription} from 'rxjs/Subscription';

@Component({
  selector: 'login',
  template: `
    <button *ngIf="!loggedIn"
            (click)="login()">
      Login
    </button>
    <button *ngIf="loggedIn"
            (click)="logout()">
      Logout
    </button>
  `
})
```

```
export class LoginComponent implements OnDestroy {
  loggedIn:boolean;
  private authChangeSubscription_: Subscription;

  constructor(private authService_:AuthService) {
    this.authChangeSubscription_ =
         authService_.authChange.subscribe(
            newAuthState => this.loggedIn
                  = (newAuthState === AuthState.LoggedIn));
  }

  login():void {
    this.authService_.login();
  }
  logout():void {
    this.authService_.logout();
  }
  ngOnDestroy() {
    this.authChangeSubscription_.unsubscribe();
  }
}
```

См. также

- Рецепт "Простое применение наблюдаемых объектов в HTTP-запросах", в котором описаны основы применения интерфейса наблюдаемых объектов.
- Рецепт "Реализация модели "издатель–подписчик" средствами класса `Subject`", в котором показано, как настроить ввод-вывод для наблюдаемых объектов из библиотеки RxJS.
- Рецепт "Создание обобщенной службы "издатель–подписчик" для замены служб `$broadcast`, `$emit` и `$on`", в котором рассматривается построение надежной модели "издатель–подписчик" для связывания отдельных компонентов приложения по каналам.
- Рецепт "Создание полноценного компонента `AutoComplete` с помощью наблюдаемых объектов" с обширным обзором некоторых служебных методов, предоставляемых в библиотеке RxJS.

Создание обобщенной службы "издатель–подписчик" для замены служб `$broadcast`, `$emit` и `$on`

Службы `$emit` и `$broadcast` действительно оказались очень полезными в Angular 1. Они давали возможность отправлять специальные события вверх и

вниз по дереву области видимости любым приемникам, которые могут ожидать подобные события. Это вынуждало разработчика действовать по очень удобному шаблону, где многие компоненты способны обмениваться событиями с центральным источником. Но применять службы `$emit` и `$broadcast` для подобных целей было совершенно неуместно, поскольку они в конечном итоге распространяли событие через огромное количество областей видимости лишь для того, чтобы достигнуть единственной намеченной цели.

В соответствующем рецепте из книги *AngularJS Web Application Development Cookbook* демонстрировалось, как создать службу "издатель-подписчик", в которой были внедрены службы **`$emit`** и **`$rootScope`**. А в рассматриваемой здесь версии данного рецепта, хотя она и несколько отличается от предыдущей, аналогичные результаты были достигнуты значительно более понятным и изящным способом.

Было бы предпочтительнее создать единственный логический объект, который может служить в качестве обобщенного канала для передачи событий от издателей к их подписчикам.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **`http://ngcookbook.herokuapp.com/2417/`**.

Подготовка

Допустим, разработка приложения начата со следующей заготовки:

```
[app/node.component.ts]

import {Component, Input} from '@angular/core';
import {PubSubService} from './publish-subscribe.service';

@Component({
  selector: 'node',
  template: `
    <p>Heard {{count}} of {{subscribeChannel}}</p>
    <button (click)="send()">Send {{publishChannel}}</button>
  `
})
export class NodeComponent {
  @Input() publishChannel:string;
  @Input() subscribeChannel:string;
  count:number = 0;

  constructor(private pubSubService_:PubSubService) {}
  send() {}
}
[app/publish-subscribe.service.ts]
```

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

@Injectable()
export class PubSubService {
  constructor() {}

  publish() {}

  subscribe() {}
}
```

Реализация

Основание для данной реализации должно быть вполне очевидно. В рассматриваемой здесь службе размещается единственный экземпляр типа `Subject`, обменивающийся событиями любого типа с данной службой через наблюдатели субъекта.

Прежде всего реализуйте приведенный ниже код, чтобы методы `subscribe()` и `publish()` выполнили все необходимые действия, когда задействуется экземпляр типа `Subject`.

[app/publish-subscribe.service.ts]

```
import {Injectable} from '@angular/core';
import {Subject} from 'rxjs/Subject';
import {Observable} from 'rxjs/Observable';
import {Observer} from 'rxjs/Observer';
import {Subscriber} from 'rxjs/Subscriber;

@Injectable()
export class PubSubService {
  private publishSubscribeSubject_:Subject<any> = new Subject();
  emitter_:Observable<any>;

  constructor() {
    this.emitter_ =
        this.publishSubscribeSubject_.asObservable();
  }

  publish(event:any):void {
    this.publishSubscribeSubject_.next(event);
  }

  subscribe(handler:NextObserver<any>):Subscriber {
    return this.emitter_.subscribe(handler);
  }
}
```

И хотя такая реализация выглядит ужасной, она решает поставленную задачу. В итоге каждое событие, издаваемое для данной службы, будет распространено всем подписчикам.

Внедрение абстракции канала

В действительности совсем не трудно ограничить издателя и подписчика таким образом, чтобы они обращали внимание только на указанный ими канал. Поэтому прежде всего внесите выделенные ниже полужирным шрифтом изменения в метод `publish()`, чтобы вложить событие в распространяемый объект.

```
[app/publish-subscribe.service.ts]

import {Injectable} from '@angular/core';
import {Subject} from 'rxjs/Subject';
import {Observable} from 'rxjs/Observable';
import {Observer} from 'rxjs/Observer';
import {Subscriber} from 'rxjs/Subscriber;

@Injectable()
export class PubSubService {
  private publishSubscribeSubject_:Subject<any> = new Subject();
  emitter_:Observable<any>;
  constructor() {
    this.emitter_ = this.publishSubscribeSubject_.asObservable();
  }

  publish(channel:string, event:any):void {
    this.publishSubscribeSubject_.next({
      channel: channel,
      event: event
    });
}

  subscribe(handler:NextObserver<any>):Subscriber {
    return this.emitter_.subscribe(handler);
  }
}
```

Благодаря этому теперь можно использовать поведение наблюдаемого объекта, чтобы ограничиться теми событиями, на которые должен обращать внимание подписчик. Для распространения событий в наблюдаемом объекте можно применить методы `filter()` и `map()`, как показано ниже. В частности, метод `filter()` возвратит новый экземпляр типа `Observable`, который распространит только то, что будет оценено как истина в его функции фильтрации. А метод `map()` возвратит новый экземпляр типа `Observable`, который преобразует все, что распространяется, в новое значение.

[app/publish-subscribe.service.ts]

```
import {Injectable} from '@angular/core';
import {Subject} from 'rxjs/Subject';
import {Observable} from 'rxjs/Observable';
import {Observer} from 'rxjs/Observer';
import {Subscriber} from 'rxjs/Subscriber;
import 'rxjs/add/operator/filter';
import 'rxjs/add/operator/map';

@Injectable()
export class PubSubService {
  private publishSubscribeSubject_:Subject<any> = new Subject();
  emitter_:Observable<any>;

  constructor() {
    this.emitter_ = this.publishSubscribeSubject_.asObservable();
  }
publish(channel:string, event:any):void {
  this.publishSubscribeSubject_.next({
    channel: channel,
    event: event
   });
}

subscribe(channel:string, handler:((value:any) => void)):
  Subscriber {
    return this.emitter_
      .filter(emission =>emission.channel === channel)
      .map(emission => emission.event)
      .subscribe(handler);
  }
}
```

Подключение компонентов к службе

Создание данной службы можно считать завершенным, но ее пока нельзя использовать в компоненте. Поэтому воспользуйтесь внедренной службой, чтобы связать компонент с каналами, указанными в его входных символьных строках.

[app/node.component.ts]

```
import {Component, Input} from '@angular/core';
import {PubSubService} from './publish-subscribe.service';

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

```
    <p>Heard {{count}} of {{subscribeChannel}}</p>
    <button (click)="send()">Send {{publishChannel}}</button>
  `
})

export class NodeComponent {
  @Input() publishChannel:string;
  @Input() subscribeChannel:string;
  count:number = 0;

  constructor(private pubSubService_:PubSubService) {}

  send() {
    this.pubSubService_
        .publish(this.publishChannel, {});
  }

  ngAfterViewInit() {
    this.pubSubService_
        .subscribe(this.subscribeChannel,
                   event => ++this.count);
  }
}
```

В качестве второго аргумента методу **publish()** передается пустой объектный литерал. Он обозначает полезную информацию для издаваемого сообщения, хотя в данном рецепте она не употребляется. Воспользуйтесь такой возможностью, если требуется отправить данные вместе с сообщением.

А теперь протестируйте свое приложение с помощью приведенного ниже компонента. В итоге вы обнаружите, что издание и подписка на каналы происходят именно так, как и предполагалось.

```
[app/root.component.ts]

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

@Component({
  selector: 'root',
  template: `
    <node subscribeChannel="foo"
          publishChannel="bar">
    </node>
    <node subscribeChannel="bar"
      publishChannel="foo">
    </node>
  `
})
export class RootComponent {}
```

Отписка от каналов

Разумеется, утечек памяти следует избегать везде, где это возможно. Для этого потребуется завершить явным образом процесс очистки после уничтожения экземпляра компонента:

```
[app/node.component.ts]

import {Component, Input, OnDestroy} from '@angular/core';
import {PubSubService} from './publish-subscribe.service';
import {Subscription} from 'rxjs/Subscription';

@Component({
  selector: 'node',
  template: `
    <p>Heard {{count}} of {{subscribeChannel}}</p>
    <button (click)="send()">Send {{publishChannel}}</button>
  `
})
export class NodeComponent implements OnDestroy {
  @Input() publishChannel:string;
  @Input() subscribeChannel:string;
  count:number = 0;
  private pubSubServiceSubscription_:Subscription;

  constructor(private pubSubService_:PubSubService) {}
  send() {
    this.pubSubService_
        .publish(this.publishChannel, {});
  }

ngAfterViewInit() {
  this.pubSubService_
      .subscribe(this.subscribeChannel,
                 event => ++this.count);
  }

  ngOnDestroy() {
    this.pubSubServiceSubscription_.unsubscribe();
  }
}
```

Принцип действия

Всякий раз, когда вызывается метод `publish()`, предоставляемое событие заключается в оболочку и передается заданным каналом центральному субъекту, который закрыт внутри службы. В то же время потребность в приеме события в другом канале при каждом вызове метода `subscribe()` вызывает определенное затруднение, поскольку без явного указания в наблюдаемом

объекте не различается, в каком именно направлении должно происходить распространение.

С помощью операций, выполняемых в методах `filter()` и `map()`, можно организовать специальное представление о распространении событий субъектом и воспользоваться этим представлением в приложении обработчика событий, наступающих в наблюдателе. Всякий раз, когда вызывается метод `subscribe()`, создается новый экземпляр типа `Observable`. Но все эти экземпляры оказываются лишь производными от одного истинного наблюдаемого объекта, которым владеет закрытый экземпляр, скрываемый внутри службы.

Дополнение

Очень важно понять, почему данная служба создана именно так, и не иначе. Важной особенностью наблюдаемых объектов является возможность составлять их в композицию. Это означает, что несколько экземпляров типа `Observable`, независимо распространяющих события, могут быть объединены в один экземпляр типа `Observable`, который будет распространять все события из объединенного источника. Этого можно добиться самыми разными способами, включая метод `flatMap()` или `merge()`. И такая возможность наблюдаемых объектов в библиотеке ReactiveX обозначается как их "способность к составлению в композицию".

Таким образом, разработчики могут найти такую способность удобной для реализации модели "издатель–подписчик", где от издателей принимаются экземпляры типа `Observable`. Они будут объединены в единый экземпляр типа `Observable`, и подписчики присоединят наблюдаемый объект к данному объединению. Что же здесь может оказаться неверным?

Соображения по поводу композиции и манипулирования наблюдаемыми объектами

Одно из главных затруднений в данном случае состоит в том, что составной наблюдаемый объект, к которому присоединяются подписчики, будет постоянно изменяться. Как и при вызове методов `map()` и `filter()`, в результате любой модификации, выполняемой над экземпляром типа `Observable`, включая композицию, возвратится новый экземпляр типа `Observable`. Этот новый экземпляр и станет тем наблюдаемым объектом, к которому присоединятся подписчики. Собственно, в этом и состоит затруднение.

Рассмотрим это затруднение поэтапно.

1. Служба "издатель–подписчик" распространяет события от наблюдаемого объекта `A`.
2. Узел `X` подписывается на данную службу и получает события от наблюдаемого объекта `A`.

3. В какой-нибудь другой части приложения наблюдаемый объект `B` добавляется в службу "издатель–подписчик".
4. Служба "издатель–подписчик" составляет наблюдаемые объекты `A` и `B` в композицию `AB`.
5. Узел `Y` подписывается на данную службу и получает события от наблюдаемого объекта `AB`.

Следует заметить, что в данном случае узел `X` будет по-прежнему принимать события только от наблюдаемого объекта `A`. Ведь это экземпляр типа `Observable`, где вызывается метод `subscribe()`.

Безусловно, для устранения рассматриваемого здесь затруднения можно предпринять ряд шагов, например, ввести дополнительный уровень абстракции между подпиской и композицией наблюдаемых объектов. Но в данный момент разумный разработчик отступит в сторону, чтобы осмыслить возникшую ситуацию. Ведь модель "издатель–подписчик" предполагается как относительно простой протокол. Это означает, что ему не следует поручать слишком много обязанностей по управлению передаваемыми событиями, как вместе с сообщениями, так и без них, чтобы заботиться больше об их доставке, а не о содержимом. В этой связи можно привести довольно сильный аргумент в пользу того, что внедрение наблюдаемых объектов в ту часть данной модели, которая относится к подписке, заметно усложнит дело.

В данном рецепте разработан изящный и простой вариант модели "издатель–подписчик", где все сложности выносятся наружу. Если же логическим объектам требуется воспользоваться издателем с наблюдаемыми объектами, решение может состоять в том, чтобы просто направить по каналу распространяемые наблюдаемыми объектами данные методу `publish()` службы "издатель–подписчик".

См. также

- Рецепт "Простое применение наблюдаемых объектов в HTTP-запросах", описывающий основы применения интерфейса наблюдаемых объектов.
- Рецепт "Реализация модели "издатель–подписчик" средствами класса `Subject`", в котором показано, как настроить ввод-вывод для наблюдаемых объектов из библиотеки RxJS.
- Рецепт "Создание наблюдаемой службы аутентификации средствами класса `BehaviorSubject`", в котором поясняется, как реактивно управлять состоянием в приложении.
- Рецепт "Создание полноценного компонента `AutoComplete` с помощью наблюдаемых объектов" с обширным обзором некоторых служебных методов, предоставляемых в библиотеке RxJS.

Отслеживание изменений в декораторе @ViewChildren средствами классов QueryList и Observable

К числу полезных видов поведения в компонентах относится возможность отслеживать в представлении изменения, происходящие в коллекциях порожденных объектов. Это довольно туманный во многих отношениях предмет, поскольку способы просмотра изменений в коллекциях многочисленны и изощренны. Правда, в Angular 2 заложено прочное основание для отслеживания подобных изменений.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/4112/**.

Подготовка

Допустим, разработка приложения начата с приведенной ниже заготовки.

```
[app/inner.component.ts]

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

@Component({
  selector: 'inner',
  template: `<p>{{val}}`
})
export class InnerComponent {
  @Input() val:number;
}

[app/outer.component.ts]

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

@Component({
  selector: 'outer',
  template: `
    <button (click)="add()">Moar</button>
    <button (click)="remove()">Less</button>
    <button (click)="shuffle()">Shuffle</button>
    <inner *ngFor="let i of list"
      val="{{i}}">
    </inner>
  `
})
export class OuterComponent {
  list:Array<number> = [];

  add():void {
```

```
    this.list.push(this.list.length)
  }

  remove():void {
    this.list.pop();
  }

  shuffle():void {
    // простая перетасовка присваивания
    this.list = this.list.sort(() =>
                (4*Math.random()>2)?1:-1);
  }
}
```

В настоящем виде данное приложение представляет собой очень простой диспетчер списка, позволяющий добавлять, удалять и перетасовывать список, вставляемый в виде экземпляров типа `InnerComponent`. Цель состоит в том, чтобы отслеживать изменения в списке и хранить ссылки на экземпляры компонентов, соответствующие представлению коллекции.

Реализация

Начните с применения декоратора `@ViewChildren` для сбора экземпляров типа `InnerComponent` в единственном списке типа `QueryList`, как выделено ниже полужирным шрифтом.

[app/outer.component.ts]

```
import {Component, ViewChildren, QueryList} from '@angular/core';
import {InnerComponent} from './inner.component';

@Component({
  selector: 'outer',
  template: `
    <button (click)="add()">Moar</button>
    <button (click)="remove()">Less</button>
    <button (click)="shuffle()">Shuffle</button>
    <inner *ngFor="let i of list"
           val="{{i}}">
    </inner>
  `
})
export class OuterComponent {
  @ViewChildren(InnerComponent) innerComponents:
    QueryList<InnerComponent>;
  list:Array<number> = [];

  add():void {
    this.list.push(this.list.length)
  }
```

```
  remove():void {
    this.list.pop();
  }

  shuffle():void {
    // простая перетасовка методом присваивания
    this.list = this.list.sort(() =>
                (4*Math.random()>2)?1:-1);
  }
}
```

Как видите, все очень просто! Теперь, когда представление типа `Outer Component` инициализировано, вы сможете воспользоваться ссылкой `this.innerComponents` для обращения к списку типа `QueryList`.

Обращение со списками типа `QueryList`

Списки типа `QueryList` являются не совсем обычными средствами вроде белых ворон в Angular 2, но, как и многие другие средства этого каркаса, они являются только соглашениями, которые придется усвоить. В данном случае они представляют собой неизменяемые и перебираемые коллекции, предоставляющие ряд методов для исследования их содержимого и его изменений.

В нашем случае наибольший интерес представляют два свойства, `last` и `changes`, экземпляра. Как и следовало ожидать, свойство `last` возвратит последний экземпляр типа `QueryList` (в данном случае — экземпляр типа `InnerComponent`, если список типа `QueryList` не пустой). А свойство `changes` возвратит экземпляр типа `Observable`, который будет распространять список типа `QueryList` всякий раз, когда в нем происходят изменения. Что касается коллекции экземпляров типа `InnerComponent`, то все операции добавления, удаления и перетасовки будут зарегистрированы как изменения.

Используя эти свойства, можно очень просто настроить представление типа `OuterComponent` на отслеживание значения, которое принял последний экземпляр типа `InnerComponent`, как показано ниже. Теперь вы сможете выяснить, что значение свойства `lastVal` по-прежнему останется актуальным при любых изменениях, инициируемых вами в коллекции типа `InnerComponent`.

```
import {Component, ViewChildren, QueryList} from '@angular/core';
import {InnerComponent} from './inner.component';

@Component({
  selector: 'app-outer',
  template: `
    <button (click)="add()">Moar</button>
    <button (click)="remove()">Less</button>
    <button (click)="shuffle()">Shuffle</button>
    <app-inner *ngFor="let i of list"
```

```
                val="{{i}}">
    </app-inner>
    <p>Value of last: {{lastVal}}</p>
  `
})
export class OuterComponent {
  @ViewChildren(InnerComponent) innerComponents:
      QueryList<InnerComponent>;
  list: Array<number> = [];
  lastVal: number;

  constructor() {}

  add() {
    this.list.push(this.list.length)
  }

  remove() {
    this.list.pop();
  }

  shuffle() {
    this.list = this.list.sort(() => (4*Math.random()>2)?1:-1);
  }

  ngAfterViewInit() {
    this.innerComponents.changes
        .subscribe(e => this.lastVal = (e.last || {}).val);
  }
}
```

Исправление ошибки в связи с изменением выражения

Если запустить рассматриваемое здесь приложение как есть, то после нажатия клавиши Moar (Больше) в первый раз появится следующее сообщение об ошибке:

```
Expression has changed after it was checked
(Выражение изменилось после его проверки)
```

Это наиболее вероятная ошибка, возникающая в Angular 2. Ее смысл довольно прост: по умолчанию вы действуете в режиме разработки, и поэтому Angular дважды проверит, не изменились ли любые граничные значения после того, как была разрешена вся логика обнаружения изменений. В данном рецепте свойство `lastVal` претерпевает изменения в связи с распространением данных из списка типа `QueryList`, что в Angular не предполагается. Следовательно, этот каркас необходимо известить явным образом, что значение может снова измениться. С этой целью можно внедрить класс

ChangeDetectorRef, позволяющий запустить цикл обнаружения изменений, как только значение изменится. Ниже показано, как это делается. В итоге все должно работать правильно и без ошибок.

```
import {Component, ViewChildren, QueryList, ngAfterViewInit,
        ChangeDetectorRef} from '@angular/core';
import {InnerComponent} from './inner.component';

@Component({
  selector: 'outer',
  template: `
    <button (click)="add()">Moar</button>
    <button (click)="remove()">Less</button>
    <button (click)="shuffle()">Shuffle</button>
    <inner *ngFor="let i of list"
           val="{{i}}">
    </inner>
    <p>Value of last: {{lastVal}}</p>
  `
})
export class OuterComponent implements AfterViewInit {
  @ViewChildren(InnerComponent) innerComponents:
      QueryList<InnerComponent>;
  list:Array<number> = [];
  lastVal:number;

  constructor(private changeDetectorRef_:ChangeDetectorRef) {}

  add():void {
    this.list.push(this.list.length)
  }

  remove():void {
    this.list.pop();
  }

  shuffle():void {
    // простая перетасовка присваивания
    this.list = this.list.sort(() =>
                (4*Math.random()>2)?1:-1);
  }

  ngAfterViewInit() {
    this.innerComponents.changes
        .subscribe(innerComponents => {
          this.lastVal = (innerComponents.last || {}).val;
          this.changeDetectorRef_.detectChanges();
        });
  }
}
```

Принцип действия

Как только представление типа `OuterComponent` будет инициализировано, вы сможете взаимодействовать со списком типа `QueryList`, полученным с помощью декоратора `@ViewChildren`. Всякий раз, когда происходят изменения в коллекции, заключаемой в оболочку списка типа `QueryList`, экземпляр типа `Observable`, доступный через его свойство `changes`, распространяет данные из списка типа `QueryList`, извещая тем самым о произошедших изменениях.

Отвращение к игроку, а не к игре

Очень важно, что обобщенный тип `Observable<QueryList>` позволяет отслеживать изменения не в массиве чисел, а в коллекции экземпляров типа `InnerComponent`. Структурная директива `ngFor` отвечает за формирование списка экземпляров типа `InnerComponent` в представлении. Именно эта коллекция, а не первоначальный массив, связана со списком типа `QueryList`.

И это совсем неплохо! Ведь декоратор `@ViewChildren` должны интересовать только компоненты, поскольку они визуализируются в представлении, а не данные, обусловившие их визуализацию именно таким образом.

Из всего сказанного выше можно сделать следующий вывод: после каждого распространения данных вполне возможно, что список типа `QueryList` окажется пустым. Как было показано выше, наблюдатель за наблюдаемым объектом в свойстве `QueryList.changes` пытается обратиться к свойству `last`, и поэтому необходимо иметь в резерве объектный литерал на тот случай, если свойство возвратит неопределенное значение (`undefined`).

См. также

- Рецепт "Простое применение наблюдаемых объектов в HTTP-запросах", в котором описаны основы применения интерфейса наблюдаемых объектов.
- Рецепт "Создание полноценного компонента `AutoComplete` с помощью наблюдаемых объектов" с обширным обзором некоторых служебных методов, предоставляемых в библиотеке RxJS.

Создание полноценного компонента `AutoComplete` с помощью наблюдаемых объектов

Наблюдаемые объекты из библиотеки RxJS предоставляют немало замечательных возможностей, и было бы непростительно упустить их. В эту библиотеку встроено немало преобразований и служебных функций, позволяющих

изящно проектировать сложные части приложения в реактивном стиле. В данном рецепте демонстрируется построение из наивной формы автозавершения надежного ряда средств, совершенствующих поведение и повышающих производительность.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/8629/**.

Подготовка

Допустим, разработка приложения начата с приведенной ниже заготовки. Цель состоит в том, чтобы значительно усовершенствовать данное приложение средствами библиотеки RxJS.

```
[app/app.module.ts]

import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {SearchComponent} from './search.component';
import {APIService} from './api.service';
import {HttpModule} from '@angular/http';

@NgModule({
  imports: [
    BrowserModule,
    HttpModule
  ],
  declarations: [
  SearchComponent
  ],
  providers: [
    APIService
  ],
  bootstrap: [
    SearchComponent
  ]
})
export class AppModule {}

[app/search.component.ts]

import {Component} from '@angular/core';
import {APIService} from './api.service';

@Component({
  selector: 'search',
  template: `
    <input #queryField (keyup)="search(queryField.value)">
    <p *ngFor="let result of results">{{result}}</p>
```

```
})
export class SearchComponent {
  results:Array<string> = [];
  constructor(private apiService_:APIService) {}
  search(query:string):void {
    this.apiService_
        .search(query)
        .subscribe(result => this.results.push(result));
  }
}

[app/api.service.ts]

import {Injectable} from '@angular/core';
import {Http} from '@angular/http';
import {Observable} from 'rxjs/Rx';

@Injectable()
export class APIService {
  constructor(private http_:Http) {}
  search(query:string):Observable<string> {
    return this.http_
           .get('static/response.json')
           .map(r => r.json()['prefix'] + query)
           // Ниже представлен лишь более изящный способ
           // случайной задержки реакции в пределах
           // от 0 до 1000 мс
           .concatMap(x =>
               Observable.of(x).delay(Math.random()*1000));
  }
}
```

Реализация

В настоящем виде данное приложение обрабатывает события от нажатия клавиш при вводе критерия поиска, выполняет HTTP-запрос статического файла в формате JSON и добавляет ответ в список результатов.

Применение наблюдаемого объекта, доступного из свойства `valueChanges` в классе `FormControl`

В каркасе Angular 2 имеется наблюдаемое поведение, уже доступное в самых разных местах. Одно из них находится в модуле `ReactiveFormsModule`, который дает возможность воспользоваться наблюдаемым объектом, присоединяемым к полю ввода данных в форме. Итак, преобразуйте это поле ввода данных, чтобы воспользоваться классом `FormControl`, предоставляющим доступ к наблюдаемому объекту в свойстве `valueChanges` данного класса, как показано ниже.

```
[app/app.module.ts]

import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {SearchComponent} from './search.component';
import {APIService} from './api.service';
import {HttpModule} from '@angular/http';
import {ReactiveFormsModule} from '@angular/forms';

@NgModule({
  imports: [
    BrowserModule,
    HttpModule,
    ReactiveFormsModule
  ],
  declarations: [
    SearchComponent
  ],
  providers: [
    APIService
  ],
  bootstrap: [
   SearchComponent
  ]
})
export class AppModule {}

[app/search.component.ts]

import {Component} from '@angular/core';
import {APIService} from './api.service';
import {FormControl} from '@angular/forms';

@Component({
  selector: 'search',
  template: `
    <input [formControl]="queryField">
    <p *ngFor="let result of results">{{result}}</p>
  `
})
export class SearchComponent {
  results:Array<string> = [];
  queryField:FormControl = new FormControl();
  constructor(private apiService_:APIService) {
    this.queryField.valueChanges.subscribe(query =>
                  this.apiService_.search(query)
                  .subscribe(result =>
                            this.results.push(result)));
  }
}
```

Устранение ложных повторений входных данных

Всякий раз, когда входное значение изменяется, Angular добросовестно инициирует запрос и обрабатывает его по мере его готовности. Так, если пользователь запросит очень длинный термин (например, *суперархиэкстраультрамегаграндиозно*), в таком случае лучше отправить единственный запрос, как только ввод данных будет завершен, вместо того, чтобы отправлять 34 запроса при каждом изменении вводимых данных.

И такая возможность встроена в наблюдаемые объекты из библиотеки RxJS. Так, в результате вызова метода `debounceTime(delay)` создается новый наблюдаемый объект типа `Observable`, который передаст только последнее значение, если в течение заданного с помощью параметра `delay` количества миллисекунд не появится никаких других значений. Вызов данного метода следует добавить к свойству `valueChanges`, поскольку оно служит источником ложных повторений, которые требуется устранить при изменении входного значения. И для этого достаточно указать задержку в течение 200 мс, как выделено ниже полужирным шрифтом.

```
[app/search.component.ts]

import {Component} from '@angular/core';
import {APIService} from './api.service';
import {FormControl} from '@angular/forms';

@Component({
  selector: 'search',
  template: `
    <input [formControl]="queryField">
    <p *ngFor="let result of results">{{result}}</p>
  `
})
export class SearchComponent {
  results:Array<string> = [];
  queryField:FormControl = new FormControl();
  constructor(private apiService_:APIService) {
    this.queryField.valueChanges
        .debounceTime(200)
        .subscribe(query =>
            this.apiService_.search(query)
                .subscribe(result =>
                    this.results.push(result)));
  }
}
```

Первоначальное происхождение термина *устранение ложных повторений* взято из области электроники, где ему соответствует термин *устранение дребезга электрических контактов*. В механических кнопках или переключателях применяются металлические контакты, замыкающие и размыкающие электрические цепи. Когда металлические контакты замыкаются, они

начинают вибрировать друг относительно друга, что на короткое время (пока система не придет в устойчивое состояние) вызывает дребезг контактов,. Подобный дребезг контактов приводит к большим помехам в электрических схемах, поскольку вызывает часто повторяющееся срабатывание кнопки или переключателя, т.е. ложное поведение. Чтобы выйти из этого затруднительного положения следует устранить дребезг контактов путем игнорирования помех, возникающих при вибрировании контактов, или введения определенной задержки перед чтением входных данных. И то и другое может быть достигнуто аппаратным или программным способом.

Игнорирование последовательных дубликатов

В данном рецепте вводимые данные читаются из текстового поля, и поэтому вполне возможно, что пользователь введет сначала один, затем другой символ и нажмет клавишу забоя. С точки зрения наблюдаемого объекта, где теперь введена задержка для устранения ложных повторений входных данных, вполне возможно, что при вводе пользователем данных в ответ будут отправлены два одинаковых значения подряд. В качестве превосходной защиты от подобных осложнений в библиотеке RxJS предоставляется выделенный ниже полужирным шрифтом метод `distinctUntilChanged()`, отвергающий распространяемые данные, если они дублируют непосредственно предшествующие им данные.

[app/search.component.ts]

```
import {Component} from '@angular/core';
import {APIService} from './api.service';
import {FormControl} from '@angular/forms';

@Component({
  selector: 'search',
  template: `
    <input [formControl]="queryField">
    <p *ngFor="let result of results">{{result}}</p>
  `
})
export class SearchComponent {
  results:Array<string> = [];
  queryField:FormControl = new FormControl();
  constructor(private apiService_:APIService) {
    this.queryField.valueChanges
        .debounceTime(200)
        .distinctUntilChanged()
        .subscribe(query =>
            this.apiService_.search(query)
                .subscribe(result =>
                    this.results.push(result)));
  }
}
```

Сведение наблюдаемых объектов

До сих пор в цепочку был связан целый ряд методов из библиотеки RxJS, и поэтому вам может показаться несколько забавным обнаружить вложенные вызовы метода `subscribe()`. Но в этом должен быть определенный смысл, поскольку в обработчике данных, распространяемых наблюдаемым объектом из свойства `valueChanges`, вызывается служебный метод, возвращающий отдельный наблюдаемый объект. В языке TypeScript он, по существу, представлен обобщенным типом `Observable<Observable<string>>`. Выглядит довольно мерзко!

Но поскольку настоящий интерес представляют лишь распространяемые символьные строки, исходящие из служебного метода, то было бы намного проще объединить все распространяемые символьные строки, исходящие из каждого возвращаемого наблюдаемого объекта, в единый наблюдаемый объект. К счастью, в библиотеке RxJS предоставляется метод `flatMap()`, упрощающий дело, сводя все данные, распространяемые из внутренних наблюдаемых объектов, в единый внешний наблюдаемый объект. В коде TypeScript с помощью `flatMap()` этот объект приводится к обобщенному типу `Observable<string>`, что, собственно, и требуется.

```
[app/search.component.ts]

import {Component} from '@angular/core';
import {APIService} from './api.service';
import {FormControl} from '@angular/forms';

@Component({
  selector: 'search',
  template: `
    <input [formControl]="queryField">
    <p *ngFor="let result of results">{{result}}</p>
  `
})
export class SearchComponent {
  results:Array<string> = [];
  queryField:FormControl = new FormControl();
  constructor(private apiService_:APIService) {
    this.queryField.valueChanges
        .debounceTime(200)
        .distinctUntilChanged()
        .flatMap(query => this.apiService_.search(query))
        .subscribe(result => this.results.push(result));
  }
}
```

Обработка неупорядоченных ответов

Если теперь вы проверите входные данные, то, безусловно, заметите, что задержка, намеренно введенная в службу программного интерфейса API, приведет к возврату ответов не по порядку. Это довольно эффективная имитация сетевой задержки, и поэтому вам придется каким-то образом решить эту проблему.

В идеальном случае было бы желательно иметь возможность отбрасывать наблюдаемые объекты, находящиеся в промежуточном состоянии, как только появится более актуальный для выполнения запрос. Допустим, пользователь ввел сначала символ `g`, а затем символ `o`. Если далее второй запрос на слово `go` появится раньше первого запроса на символ `g`, то последний было бы желательно отбросить и забыть о нем, поскольку он уже неактуален.

И эта задача упрощается с помощью выделенного ниже полужирным шрифтом метода `switchMap()` из библиотеки RxJS. Этот метод делает то же самое, что и метод `flatMap()`, но он отпишется от любых промежуточных наблюдаемых объектов, не распространивших еще никаких данных. В итоге ложные повторения входных данных в компоненте `AutoComplete` будут устранены, а избыточные запросы и возвращаемые по порядку результаты — проигнорированы.

```
[app/search.component.ts]

import {Component} from '@angular/core';
import {APIService} from './api.service';
import {FormControl} from '@angular/forms';

@Component({
  selector: 'search',
  template: `
    <input [formControl]="queryField">
    <p *ngFor="let result of results">{{result}}</p>
  `
})
export class SearchComponent {
  results:Array<string> = [];
  queryField:FormControl = new FormControl();
  constructor(private apiService_:APIService) {
    this.queryField.valueChanges
        .debounceTime(200)
        .distinctUntilChanged()
        .switchMap(query => this.apiService_.search(query))
        .subscribe(result => this.results.push(result));
  }
}
```

Принцип действия

В данном рецепте немало подвижных составляющих, но главный его предмет остается неизменным: наблюдаемые объекты из библиотеки RxJS предоставляют немало методов, способных распространять по каналу выходные данные от одного наблюдаемого объекта к совершенно другому наблюдаемому объекту. С их помощью можно также объединить несколько наблюдаемых объектов в единое целое и добавить зависящие от состояния операции в поток входных данных. К концу данного рецепта истинный потенциал реактивного программирования должен стать для вас вполне очевидным.

См. также

- Рецепт "Простое применение наблюдаемых объектов в HTTP-запросах", в котором демонстрируются основы применения интерфейса наблюдаемых объектов.
- Рецепт "Реализация модели "издатель–подписчик" средствами класса `Subject`", в котором описано, как настроить ввод-вывод для наблюдаемых объектов из библиотеки RxJS.
- Рецепт "Создание наблюдаемой службы аутентификации средствами класса `BehaviorSubject`", поясняющий, как реактивно управлять состоянием в приложении.
- Рецепт "Создание обобщенной службы "издатель–подписчик" для замены служб `$broadcast`, `$emit` и `$on`", в котором рассматривается построение надежной модели "издатель–подписчик" для связывания отдельных компонентов приложения по каналам.

Глава 6

Модуль Component Router

В этой главе даются следующие рецепты.

- Настройка приложения на поддержку простых маршрутов
- Навигация по ссылкам с помощью директивы `routerLink`
- Навигация с помощью службы `Router`
- Выбор стратегии обнаружения ресурсов типа `LocationStrategy` для построения пути
- Реализация сохраняющего состояние маршрута с помощью директивы `routerLinkActive`
- Реализация вложенных представлений с помощью параметров маршрута и порожденных маршрутов
- Обработка матричных параметров URL и массивов маршрутизации
- Добавление элементов управления аутентификацией маршрутов с помощью ограничителей маршрутов

Введение

В некоторых средствах Angular 2 следует усматривать нечто большее, чем маршрутизатор компонента из модуля Component Router. В новой реализации механизма маршрутизации предоставляется целый ряд замечательных средств, отсутствовавших в Angular 1, где ощущалась их серьезная нехватка.

В каркасе Angular 2 реализуются матричные параметры. Это совершенно новый синтаксис для структур URL, основанный на точках с запятой и первоначально предложенный Тимом Бернерсом-Ли (Tim Berners-Lee) в 1996 году. Он дает возможность надежно связывать параметры не только с отдельным URL, но и с разными уровнями в этом URL. Теперь в URL можно вводить дополнительное измерение для отслеживания состояния веб-приложения.

Кроме того, модуль Component Router предоставляет возможность изящно вкладывать представления друг в друга, а также определять без труда маршруты и ссылки на иерархии подобных компонентов. Это означает, что вы можете извлечь наибольшую пользу из своих приложений, определив их как независимые модули.

И, наконец, в модуль Component Router полностью интегрированы структуры наблюдаемых объектов. Он предоставляет прекрасную возможность для навигации и контроля над ней в приложении.

Настройка приложения на поддержку простых маршрутов

Главной особенностью поведения одностраничных веб-приложений является способность выполнять навигацию без формальной перезагрузки страницы в браузере. Каркас Angular 2 оснащен достаточными средствами, чтобы обойти это стандартное поведение браузера и определить такую структуру маршрутизации, которая придаст вид фактической навигации страниц.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/6214/**.

Подготовка

Допустим, имеется приведенная ниже глобально определенная функция. Цель состоит в том, чтобы дать возможность перемещаться в окне браузера из сценария JavaScript, используя интерфейс HTML5 History API.

```
function visit(uri) {
  // В данном рецепте состояние или заголовок
  // страницы не имеет значения
  window.history.pushState(null, null, uri);
}
```

Реализация

Чтобы смоделировать навигацию по страницам в браузере средствами Angular, необходимо предпринять несколько шагов с целью создать доступное для перемещения одностраничное приложение.

Установка базового URL

В качестве первого шага настройки приложения следует указать базовый URL. Тем самым браузер получает инструкцию, какие именно сетевые запросы ему следует делать при получении относительных URL.

Всякий раз, когда на странице делается запрос с использованием относительного URL для генерирования сетевого запроса, выбирается текущий домен, к которому затем добавляется относительный URL. Знак "/", предваряющий относительный URL, означает, что браузер должен использовать корневой каталог. Если же этот знак не указан, то к относительному пути будет добавляться то, что указано в базовой гиперссылке. Такое поведение проявляется повсюду в URL, включая запрашиваемые статические ресурсы, точки привязки гиперссылок (anchor links) и интерфейс API для ведения предыстории.

Дескриптор **<base>** является не частью Angular, а элементом разметки стандарта HTML5.

Ниже приведены некоторые характерные тому примеры.

```
[Пример 1]
// <base href="/">
// Первоначальное местоположение страницы: foo.com

visit('bar');
// Новое местоположение страницы: foo.com/bar

visit('bar');
// Новое местоположение страницы: foo.com/bar
// Браузер распознает этот путь как относительный без
// предваряющего знака /, и поэтому он перейдет на страницу
// с той же "глубиной", что и прежде

visit('bar/');
// Новое местоположение страницы: foo.com/bar/
// То же, что и прежде, но завершающий знак / окажется важным,
// если сделать тот же самый вызов еще раз

visit('bar/');
// Новое местоположение страницы: foo.com/bar/bar/
// Браузер распознает, что этот URL оканчивается знаком /,
// и поэтому переход по данному пути будет рассматриваться
// как навигация по подчиненному пути

visit('/qux');
// Новое местоположение страницы: foo.com/qux
// Если URL предваряется знаком /, браузер
// интерпретирует его как необходимость произвести
// навигацию, начиная с корня

[Пример 2]
// <base href="xyz/">
// Первоначальное местоположение страницы: foo.com
```

```
visit('bar');
// Новое местоположение страницы: foo.com/xyz/bar
// К базовому URL добавляется относительный URL

visit('bar');
// Новое местоположение страницы: foo.com/xyz/bar
// Как и прежде, локальный путь интерпретируется
// браузером таким же образом

visit('/qux');
// Новое местоположение страницы: foo.com/qux
// Следует заметить, что в данном случае указан
// относительный путь, происходящий из корневого
// каталога, и поэтому базовая гиперссылка игнорируется
```

Определение маршрутов

Далее требуется определить конкретные маршруты в приложении. Но в данном рецепте важнее раскрыть особенности настройки маршрутизации, чем определения маршрутов и их взаимной навигации. Поэтому достаточно определить пока что единственный универсальный маршрут.

Нетрудно догадаться, что представления маршрутов в Angular 2 определяются как компоненты. Путь по каждому маршруту представлен, по крайней мере, символьной строкой, с которой должно совпасть местоположение искомой страницы в браузере, а также компонентом, который ему соответствует. Это можно сделать с помощью объекта, реализующего интерфейс `Routes`, представляющий массив определений маршрутов.

Целесообразно, чтобы определения маршрутов происходили очень рано в процессе инициализации приложения, и это можно сделать в определении модуля верхнего уровня. С этой целью создайте прежде всего компонент представления, к которому приводит данный маршрут:

```
[app/default.component.ts]

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

@Component({
  template: 'Default component!'
})
export class DefaultComponent {}
```

Далее импортируйте модуль `RouterModule` и интерфейс `Routes` (в частности, компонент `DefaultComponent`) везде, где определяется модуль приложения, а также определите универсальный маршрут в массиве типа `Routes` следующим образом:

```
[app/app.module.ts]

import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {RouterModule, Routes} from '@angular/router';
import {RootComponent} from './root.component';
import {DefaultComponent} from './default.component';

const appRoutes:Routes = [
  {path: '**', component: DefaultComponent}
];

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

Предоставление маршрутов приложению

Выше маршруты были определены в объекте, но приложению по-прежнему неизвестно об их существовании. Чтобы ему стало известно о них, можно воспользоваться выделенным ниже полужирным шрифтом методом `forRoot()`, определенным в модуле `RouterModule`. Этот метод выполняет всю черновую работу по установке маршрутов в приложении, а также передает ряд поставщиков маршрутизации для применения повсюду в приложении. После этого ваше приложение будет полностью готово обрабатывать определенные вами маршруты.

```
[app/app.module.ts]

import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {RouterModule, Routes} from '@angular/router';
import {RootComponent} from './root.component';
import {DefaultComponent} from './default.component';

const appRoutes:Routes = [
  {path: '**', component: DefaultComponent}
];
@NgModule({
```

```
  imports: [
    BrowserModule,
    RouterModule.forRoot(appRoutes)
  ],
  declarations: [
    DefaultComponent,
    RootComponent
  ],
  bootstrap: [
    RootComponent
  ]
})
export class AppModule {}
```

Визуализация компонентов маршрутов с помощью директивы routerOutlet

Для визуализации компонента требуется определить место, которое в Angular 2 принимает форму директивы routerOutlet. Эта директива указывается в компоненте, присоединяемым к активному маршруту, и в ней будет визуализироваться данный компонент. Ради простоты в данном рецепте можно воспользоваться этой директивой в корневом компоненте приложения следующим образом:

```
[app/app.component.ts]

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

@Component({
  selector: 'root',
  template: `
    <h1>Root component</h1>
    <router-outlet></router-outlet>
  `
})
export class RootComponent {}
```

Вот, собственно, и все! Теперь в вашем приложении определен единственный маршрут, по которому компонент DefaultComponent будет визуализирован в компоненте RootComponent.

Принцип действия

В данном рецепте не демонстрируется усложненный пример маршрутизации, поскольку всякий маршрут, по которому можно перейти, приведет к тому же самому компоненту. Тем не менее в данном рецепте демонстрируются следующие основополагающие принципы маршрутизации в Angular.

- В своей простейшей форме маршрут состоит из строкового пути, совпадающего с путем в браузере, и компонента, который должен быть визуализирован при активизации данного маршрута.
- Маршруты устанавливаются через модуль `RouterModule`. В данном примере используется единственный модуль, и поэтому установить маршрут можно лишь один раз с помощью метода `forRoot()`. Следует, однако, иметь в виду, что структуру маршрутизации можно разделить на части и между разными модулями, определяемыми с помощью декоратора `@NgModule`.
- Навигация по маршруту приведет к визуализации одного компонента в другом компоненте везде, где существует дескриптор `<router-outlet>`. Имеется немало способов для настройки и усложнения такого поведения, но ради простоты в данном рецепте эти способы не демонстрируются на примере единственного модуля.

Дополнение

При выполнении приложений Angular 2 без какой-либо формы маршрутизации не возникнет никаких осложнений. Если же в приложении не требуется распознавать URL и манипулировать им, то файлы и модули маршрутизации из этого приложения могут быть полностью удалены.

Первоначальная загрузка страницы

Предполагаемая последовательность действий пользователей может выглядеть следующим образом.

1. Пользователь посещает страницу по адресу `http://www.foo.com/`.
2. Сервер находит соответствие пустого маршрута с файлом `index.html`.
3. Загружается страница и запрашиваются статические файлы.
4. Загружаются статические файлы Angular и приложение запускается на выполнение.
5. Пользователь щелкает на ссылках и перемещается по сайту.
6. Все работает, как и предполагалось, поскольку Angular полностью управляет навигацией и маршрутизацией.

Это, конечно, идеальный случай. Поэтому рассмотрим другой случай, где выполняются следующие действия.

1. Пользователь уже посетил страницу по адресу `http://www.foo.com/` и сделал на нее закладку.

2. Пользователь вводит путь `foo.com/bar` в поле веб-адреса и непосредственно переходит по нему.
3. Сервер распознает путь запроса как `/bar` и пытается обработать запрос.

В зависимости от конкретной конфигурации сервера это может вызвать определенные осложнения. Дело в том, что когда пользователь посещал страницу по пути `foo.com/bar`, то никакого запроса этого ресурса на сервер не поступило, поскольку в Angular было лишь смоделировано настоящее событие навигации.

Данный случай обсуждается подробнее в другом месте этой книги. Следует, однако, иметь в виду, что если сервер не сконфигурирован правильно, то во втором случае пользователь может обнаружить сообщение с кодом 404 ошибки на странице вместо вашего приложения.

См. также

- Рецепт "Навигация по ссылкам с помощью директивы `routerLink`", в котором демонстрируется, каким образом осуществляется навигация по приложениям Angular.
- Рецепт "Навигация с помощью службы `Router`", где служба Angular применяется для навигации по приложению.
- Рецепт "Реализация сохраняющего состояние маршрута с помощью директивы `routerLinkActive`", в котором показано, как интегрировать поведение приложения с сохранением состояния URL.

Навигация по ссылкам с помощью директивы `routerLink`

Навигация по одностраничному веб-приложению является основополагающей задачей. И для ее решения в Angular предоставляется встроенная директива `routerLink`.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **`http://ngcookbook.herokuapp.com/9983/`**.

Подготовка

Начните с установки приложения, написанного в предыдущем рецепте. Цель состоит в том, чтобы добавить в данное приложение еще один маршрут вместе с компонентом, а также предоставить возможность перемещаться по гиперссылкам.

Принцип действия

Прежде всего создайте еще один компонент ArticleComponent и связанный с ним маршрут следующим образом:

```
 [app/article/article.component.ts]

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

@Component({
  template: 'Article component!'
})
export class ArticleComponent {}
```

Задайте маршрут к статье вместе с новым компонентом, как показано ниже.

```
[app/app.module.ts]

import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {RouterModule, Routes} from '@angular/router';
import {RootComponent} from './root.component';
import {DefaultComponent} from './default.component';
import {ArticleComponent} from './article.component';

const appRoutes:Routes = [
  {path: 'article', component: ArticleComponent},
  {path: '**', component: DefaultComponent}
];

@NgModule({
  imports: [
    BrowserModule,
    RouterModule.forRoot(appRoutes)
  ],
  declarations: [
    DefaultComponent,
    ArticleComponent,
    RootComponent
  ],
  bootstrap: [
    RootComponent
  ]
})
export class AppModule {}
```

Определив маршруты, вы можете теперь создать элементарную панель навигации, состоящую из гиперссылок по директивам routerLink. Разметка в окружении дескриптора <router-outlet> останется независимой от

маршрута. Поэтому корневой компонент приложения, по-видимому, оказывается подходящим местом для навигационных ссылок.

Директива `routerLink` доступна как часть модуля `RouterModule`. Поэтому можете перейти непосредственно к добавлению ряда дескрипторов привязки, как выделено ниже полужирным шрифтом.

```
[app/root.component.ts]

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

@Component({
  selector: 'root',
  template: `
    <h1>Root component</h1>
    <a [routerLink]="''">Default</a>
    <a [routerLink]="'article'">Article</a>
    <router-outlet></router-outlet>
  `
})
export class RootComponent {}
```

В данном случае маршруты просты и статичны, поэтому допускается привязка к символьной строке по директиве `routerLink`, где принимается также обозначение массива:

```
[app/root.component.ts]

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

@Component({
  selector: 'root',
  template: `
    <h1>Root component</h1>
    <a [routerLink]="['']">Default</a>
    <a [routerLink]="['article']">Article</a>
    <router-outlet></router-outlet>
  `
})
export class RootComponent {}
```

По замыслу данного рецепта в обозначении массива ничего не указано. Но при разработке более сложных структур URL обозначение массива становится удобным, поскольку оно позволяет формировать ссылки по частям.

Принцип действия

В общем, рассматриваемая здесь реализация ничем особенным не отличается от поведения обыкновенного атрибута разметки `href`. В конечном счете

маршруты ведут себя и структурированы таким же самым образом. Но важное отличие данной реализации заключается в том, что применение в ней директивы routerLink вместо атрибута href позволяет перемещаться по приложению в стиле Angular, не вынуждая браузер интерпретировать щелчок кнопкой мыши на дескрипторе привязки как навигационное событие в стиле Angular.

Дополнение

Безусловно, директива routerLink более совершенна, поскольку она допускает большее расширение как инструментальное средство для навигации. Но в конечном счете это атрибут HTML-разметки, и поэтому ничто не мешает присоединить директиву routerLink, например, к экранной кнопке, как выделено ниже полужирным шрифтом.

```
[app/root.component.ts]

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

@Component({
  selector: 'root',
  template: `
    <h1>Root component</h1>
    <button [routerLink]="['']">Default</button>
    <button [routerLink]="['article']">Article</button>
    <router-outlet></router-outlet>
  `
})
export class RootComponent {}
```

Более того, можно заметить, что обозначение массива позволяет динамически формировать ссылки через все многочисленные средства привязки данных, предоставляемые в Angular, как показано ниже. А по мере совершенствования структуры URL становится ясно, каким образом разумная привязка данных в приложении способствует весьма изящному формированию динамических ссылок.

```
app/root.component.ts]

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

@Component({
  selector: 'root',
  template: `
    <h1>Root component</h1>
    <a [routerLink]="[defaultPath]">Default</a>
    <a [routerLink]="[articlePath]">Article</a>
    <router-outlet></router-outlet>
  `
```

```
})
export class RootComponent {
  defaultPath:string = '';
  articlePath:string = 'article';
}
```

Соображения об упорядочении маршрутов

Упорядочение маршрутов в определении интерфейса `Routes` обозначает приоритетность каждого из них по убыванию. Допустим, что в примере из данного рецепта требуется изменить на обратный порядок следования маршрутов, как выделено ниже полужирным шрифтом.

[app/app.module.ts]

```
import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {RouterModule, Routes} from '@angular/router';
import {RootComponent} from './root.component';
import {DefaultComponent} from './default.component';
import {ArticleComponent} from './article.component';

const appRoutes:Routes = [
  {path: '**', component: DefaultComponent},
  {path: 'article', component: ArticleComponent}
];

@NgModule({
  imports: [
    BrowserModule,
    RouterModule.forRoot(appRoutes)
  ],
  declarations: [
    DefaultComponent,
    ArticleComponent,
    RootComponent
  ],
  bootstrap: [
    RootComponent
  ]
})
export class AppModule {}
```

Экспериментируя с маршрутами, вы непременно обнаружите, что URL в браузере изменяется правильно при различных взаимодействиях по директиве `routerLink`, но в обоих маршрутах компонент `DefaultComponent` используется для визуализации представления. Это происходит просто потому, что все маршруты совпадают с универсальным шаблоном **, и как только в Angular будет обнаружен совпавший маршрут, обходить остальные

маршруты не имеет смысла. Эту особенность следует иметь в виду при составлении крупных таблиц маршрутов.

См. также

- Рецепт "Настройка приложения на поддержку простых маршрутов" с описанием основ маршрутизации в Angular.
- Рецепт "Навигация с помощью службы `Router`", где служба Angular применяется для навигации по приложению.
- Рецепт "Реализация сохраняющего состояние маршрута с помощью директивы `routerLinkActive`", в котором показано, как интегрировать поведение приложения с сохранением состояния URL.
- Рецепт "Реализация вложенных представлений с помощью параметров маршрута и порожденных маршрутов", в котором приводится пример, демонстрирующий порядок настройки URL в Angular для поддержки вложения и передачи данных.
- Рецепт "Обработка матричных параметров URL и массивов маршрутизации", в котором показана поддержка в Angular матричных параметров URL.

Навигация с помощью службы `Router`

Директиву `routerLink` можно применять для навигации не только в шаблоне, но и в сценарии JavaScript. Для этой цели в службе `Router` предоставляется метод `navigate()`.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **`http://ngcookbook.herokuapp.com/8004/`**.

Подготовка

Начните с версии приложения, полученной в конце раздела "Реализация" из рецепта "Навигация по ссылкам с помощью директивы `routerLink`". Цель состоит в том, чтобы добавить в данное приложение дополнительный маршрут вместе с компонентом, а также предоставить возможность перемещаться по гиперссылкам.

Принцип действия

Вместо применения директивы `routerLink`, что было бы наиболее благоразумно в данном случае, процесс навигации можно также запустить, используя службу `Router`. Прежде всего добавьте экранные кнопки навигации

и присоедините к ним пустые обработчики событий от щелчков кнопками мыши следующим образом:

```
[app/root.component.ts]

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

@Component({
  selector: 'root',
  template: `
    <h1>Root component</h1>
    <button (click)="visitDefault()">Default</button>
    <button (click)="visitArticle()">Article</button>
    <router-outlet></router-outlet>
  `
})
export class RootComponent {
  visitDefault():void {}
  visitArticle():void {}
}
```

Импортируйте службу `Router` и воспользуйтесь методом `navigate()`, чтобы изменить местоположение страницы, как выделено ниже полужирным шрифтом. Благодаря этому дополнению можно перемещаться по данному приложению таким же образом, как и прежде.

```
[app/root.component.ts]

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

@Component({
  selector: 'root',
  template: `
    <h1>Root component</h1>
    <button (click)="visitDefault()">Default</button>
    <button (click)="visitArticle()">Article</button>
    <router-outlet></router-outlet>
  `
})
export class RootComponent {
  constructor(private router:Router) {}
  visitDefault():void {
    this.router.navigate(['']);
  }
  visitArticle():void {
    this.router.navigate(['article']);
  }
}
```

Принцип действия

Служба `Router` предоставляет интерфейс API, с помощью которого можно, среди прочего, управлять поведением приложения. В частности, ее методу `navigate()` передается в качестве параметра маршрут, структурированный в виде массива и действующий аналогично массивам типа `Array`, привязанным к директиве `routerLink`.

Дополнение

Очевидно, что это совершенный антишаблон для написания приложений, предполагающих масштабирование. В данном случае директива `routerLink` оказывается намного более лаконичным и эффективным вариантом выбора для создания простой панели навигации. Тем не менее служба `Router` является в равной степени эффективным средством для обхода структуры маршрутов в приложении Angular.

См. также

- Рецепт "Навигация по ссылкам с помощью директивы `routerLink`", в котором демонстрируется, каким образом осуществляется навигация по приложениям Angular.
- Рецепт "Реализация сохраняющего состояние маршрута с помощью директивы `routerLinkActive`", в котором описано, как интегрировать поведение приложения с сохранением состояния URL.
- Рецепт "Обработка матричных параметров URL и массивов маршрутизации", где демонстрируется поддержка в Angular матричных параметров URL.
- Рецепт "Добавление элементов управления аутентификацией маршрутов с помощью ограничителей маршрутов" с подробным описанием всего процесса настройки защищенных маршрутов в приложении.

Выбор стратегии обнаружения ресурсов типа `LocationStrategy` для построения пути

Для приложения необходимо сделать простой, но важный выбор типа `LocationStrategy` для применяемой стратегии обнаружения ресурсов. Выбранному типу `LocationStrategy` соответствуют следующие URL:

- `PathLocationStrategy`: `foo.com/bar`
- `HashLocationStrategy`: `foo.com/#/bar`

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/1355/**.

Реализация

По умолчанию в Angular 2 выбирается тип `PathLocationStrategy` стратегии обнаружения. Если же требуется выбрать тип `HashLocationStrategy`, его можно импортировать из модуля `@angular/common`. После импорта его можно указать в качестве поставщика в объектном литерале следующим образом:

```
[app/app.module.ts]

import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {RouterModule, Routes} from '@angular/router';
import {RootComponent} from './root.component';
import {DefaultComponent} from './default.component';
import {ArticleComponent} from './article.component';
import {LocationStrategy, HashLocationStrategy}
        from '@angular/common';

const appRoutes:Routes = [
  {path: 'article', component: ArticleComponent},
  {path: '**', component: DefaultComponent},
];
@NgModule({
  imports: [
    BrowserModule,
    RouterModule.forRoot(appRoutes)
  ],
  declarations: [
    DefaultComponent,
    ArticleComponent,
    RootComponent
  ],
  providers: [
    {provide: LocationStrategy, useClass: HashLocationStrategy}
  ],
  bootstrap: [
    RootComponent
  ]
})
export class AppModule {}
```

Благодаря этому дополнению данное приложение осуществит переход к префиксу **#/** во всех определяемых в нем URL. И это произойдет согласованно с остальной частью приложения, где можно, как правило, пользоваться определениями маршрутизации, не беспокоясь о добавлении префикса **#/**.

Дополнение

У каждой из рассматриваемых здесь стратегий обнаружения имеются свои компромиссы. Как отмечается в документации по Angular, выбрав одну из них, не рекомендуется переходить к другой, поскольку закладки, поисковая оптимизация и предыстория пользователей будут связаны со стратегией обнаружения ресурсов по URL, применяемой при посещении веб-страницы.

Особенности стратегии обнаружения ресурсов типа `PathLocationStrategy`:

- в этом случае URL предоставляются конечному пользователю в обычном виде;
- сервер должен быть сконфигурирован для обработки загружаемых страниц по любому пути в приложении;
- это позволяет выполнять на сервере гибридную визуализацию маршрутов ради повышения производительности.

Особенности стратегии обнаружения ресурсов типа `HashLocationStrategy`:

- в этом случае URL могут выглядеть необычно для конечного пользователя;
- никакой конфигурации не требуется, если в корневом каталоге присутствует файл `index.html`. Это удобный выбор, если требуется обслужить лишь статические файлы (например, сайт компании Amazon на основе платформы AWS).
- Нелегко сочетается с гибридной визуализацией маршрутов на сервере.

Конфигурирование сервера приложений для применения стратегии обнаружения ресурсов типа `PathLocationStrategy`

Каркас Angular обладает достаточно развитой логикой, чтобы распознать состояние браузера и соответственно управлять им после начальной загрузки. Но для начальной загрузки требуется сначала загрузить в браузер статически скомпилированные ресурсы JavaScript, которые затем загрузят Angular. При первом посещении пользователем корневого домена (например, `foo.com`) сервер, как правило, отправляет в ответ файл `index.html`, который, в свою очередь, запросит статические ресурсы во время визуализации страницы в окне браузера. И таким образом Angular вступит в действие!

Но в тех случаях, когда пользователь посещает сначала ресурс не по корневому пути (например, `foo.com/bar`), браузер отправит запрос на сервер по пути `foo.com/bar`. Если некорректно настроить сервер, это приведет к типичной ошибке, когда файл `index.html` будет возвращаться только по корневому пути `foo.com`.

Чтобы стратегия обнаружения ресурсов типа `PathLocationStrategy` действовала правильно в любом случае, необходимо сконфигурировать

веб-сервер таким образом, чтобы установить универсальный маршрут для всех запросов, содержащих пути, предназначенные для маршрутизации одностраничного приложения на стороне клиента, а также для неизменного возврата файла `index.html`. Иными словами, при первоначальном посещении страницы по пути `foo.com`, `foo.com/bar` или `foo.com/bar/baz` в браузере, так или иначе, возвратится один и тот же файл `index.html`. А при последующей начальной загрузке Angular запросит текущий путь в браузере, распознав свой путь и представление, которое требуется визуализировать.

Реализация сохраняющего состояние маршрута с помощью директивы `routerLinkActive`

При разработке приложений нередко требуется создать средства, зависящие от текущей страницы, на которой работает приложение. Однократная проверка положения в браузере не вызывает особых затруднений, поскольку средства Angular и стандартные интерфейсы API в браузерах позволяют легко узнать URL текущей страницы.

Но дело несколько усложняется, когда требуется, чтобы состояние страницы отражало состояние URL. Например, в том случае, если требуется наглядно показать, какая ссылка соответствует текущей странице. Чтобы решить такую задачу сызнова, потребуется своего рода конечный автомат, которому должно быть известно, когда происходят навигационные события и какие изменения следует произвести по каждому отдельному маршруту. К счастью, в Angular 2 предоставляется ряд превосходных инструментальных средств, готовых к решению подобной задачи.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **`http://ngcookbook.herokuapp.com/3308/`**.

Подготовка

Начните с создания массива типа `Array` и реализации, основанной на дескрипторах привязки и представленной ранее в рецепте "Навигация по ссылкам с помощью директивы `routerLink`". Цель состоит в том, чтобы воспользоваться директивой `routerLinkActive` для внедрения простого поведения, сохраняющего состояние маршрута.

Реализация

Директива `routerLinkActive` позволяет применять другие классы при определенных условиях, когда текущий маршрут совпадает с соответствую-

щей директивой routerLink в том же самом элементе разметки. Итак, продолжите, добавив ее как директиву атрибута в каждую ссылку, а также соответствующий CSS-класс, как выделено ниже полужирным шрифтом.

```
[app/root.component.ts]

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

@Component({
  selector: 'root',
  template: `
    <h1>Root component</h1>
    <a [routerLink]="['']"
      [routerLinkActive]="'active-navlink'">Default</a>
    <a [routerLink]="['article']"
      [routerLinkActive]="'active-navlink'">Article</a>
    <router-outlet></router-outlet>
  `,
  styles: [`
    .active-navlink {
      color: red;
      text-transform: uppercase;
    }
  `]
})
export class RootComponent {}
```

Вот, собственно, и все, что требуется для того, чтобы активизировать ссылки! Обратите внимание, что Angular применит при определенных условиях CSS-класс active-navlink, исходя из текущего маршрута.

Но если протестировать приведенный выше исходный код, то можно заметить, что обе ссылки активизируются по маршруту /article. Это объясняется тем, что по умолчанию все директивы routerLink, совпадающие с текущим маршрутом, помечаются средствами Angular как активные.

Такое поведение оказывается удобным в тех случаях, когда иерархию ссылок требуется отобразить как активную. Например, в маршруте **/user/123/detail** было бы целесообразно показать все ссылки **/user**, **/user/123** и **/user/123/detail** как активные.

Но в данном рецепте пользы от такого поведения не особенно много. Впрочем, в каркасе Angular предоставляется еще одна директива маршрутизации, которая называется outerLinkActiveOptions и привязывается к объекту вариантов выбора options. В данном случае пригодится свойство exact объекта options, определяющее, следует ли применять активное состояние только при точном совпадении, как показано ниже. Теперь вы обнаружите, что каждая ссылка активизируется лишь по соответствующему маршруту.

```
[app/root.component.ts]

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

@Component({
  selector: 'root',
  template: `
    <h1>Root component</h1>
    <a [routerLink]="['']"
      [routerLinkActive]="'active-navlink'"
      [routerLinkActiveOptions]="{exact:true}">Default</a>
    <a [routerLink]="['article']"
      [routerLinkActive]="'active-navlink'"
      [routerLinkActiveOptions]="{exact:true}">Article</a>
    <router-outlet></router-outlet>
  `,
  styles: [`
    .active-navlink {
      color: red;
      text-transform: uppercase;
    }
  `]
})
export class RootComponent {}
```

Принцип действия

В реализации директивы `routerLinkActive` производится подписка на события, наступающие при изменениях в навигации и распространяемые средствами Angular из службы `Router`. Обнаружив событие `NavigationEnd`, эта директива выполнит обновление всех присоединенных дескрипторов HTML-разметки, включая ввод и удаление применяемых "активных" CSS-классов, к которым привязан элемент разметки через данную директиву.

Дополнение

Если директиву `routerLinkActive` требуется привязать к динамическому значению, то предыдущий синтаксис позволяет добиться именно этого. Например, член компонента можно привязать и модифицировать где угодно, а все остальное будет сделано в Angular автоматически. Но если этого не требуется, то Angular обработает директиву `routerLinkActive` без квадратных скобок, обозначающих привязку данных. В данном случае значение директивы совсем не обязательно должно быть выражением Angular, а следовательно, можно удалить вложенные кавычки.

Приведенный ниже код ведет себя аналогичным образом.

```
[app/root.component.ts]

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

@Component({
  selector: 'root',
  template: `
    <h1>Root component</h1>
    <a [routerLink]="['']"
      routerLinkActive="active-navlink"
      [routerLinkActiveOptions]="{exact:true}">
      Default</a>
    <a [routerLink]="['article']"
       routerLinkActive="active-navlink"
       [routerLinkActiveOptions]="{exact:true}">
       Article</a>
    <router-outlet></router-outlet>
  `,
  styles: [`
     .active-navlink {
       color: red;
       text-transform: uppercase;
     }
  `]
})
export class RootComponent {}
```

См. также

- Рецепт "Настройка приложения на поддержку простых маршрутов" с описанием основ маршрутизации в Angular.
- Рецепт "Навигация по ссылкам с помощью директивы `routerLink`", в котором показано, каким образом осуществляется навигация по приложениям Angular.
- Рецепт "Реализация сохраняющего состояние маршрута с помощью директивы `routerLinkActive`" с примером, демонстрирующим порядок настройки URL в Angular для поддержки вложения и передачи данных.
- Рецепт "Реализация вложенных представлений с помощью параметров маршрута и порожденных маршрутов", в котором приводится пример, демонстрирующий порядок настройки URL в Angular для поддержки вложения и передачи данных.
- Рецепт "Добавление элементов управления аутентификацией маршрутов с помощью ограничителей маршрутов" с подробным описанием всего процесса настройки защищенных маршрутов в приложении.

Реализация вложенных представлений с помощью параметров маршрута и порожденных маршрутов

В маршрутизаторе компонентов Angular 2 реализована концепция порожденных маршрутов. Как и следовало ожидать, это приводит к понятию рекурсивно определяемых представлений в таблице, что дает возможность очень удобно и изящно написать приложение.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/7892/**.

Подготовка

Начните с создания массива типа `Array` и реализации, основанной на дескрипторах привязки и представленной ранее в рецепте "Навигация по ссылкам с помощью директивы `routerLink`". Цель состоит в том, чтобы расширить данное простое приложение и включить в него путь `/article` к списочному представлению статей, а также путь `/article/:id` к их подробному представлению.

Реализация

Прежде всего модифицируйте структуру маршрутов для данного простого приложения, расширив путь `/article` составляющими `/` и `/:id`. Маршруты определяются иерархически, причем у каждого могут быть свои маршруты, порожденные от него с помощью свойства `children`.

Добавление цели маршрутизации в родительский компонент

Сначала необходимо модифицировать существующий компонент `ArticleComponent`, чтобы он мог содержать порожденные представления. Как и следовало ожидать, порожденное представление визуализируются точно так же, как и из корневого компонента с помощью директивы `routerOutlet`, выделенной ниже полужирным шрифтом. И хотя это пока еще ничего особенного не дает, но добавлением директивы `routerOutlet` описывается, каким образом иерархии компонентов должны быть визуализированы средствами Angular.

```
[app/article.component.ts]

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

@Component({
```

```
  template: `
    <h2>Article</h2>
    <router-outlet></router-outlet>
  `
})
export class ArticleComponent {}
```

Определение вложенных порожденных представлений

В данном рецепте было бы желательно, чтобы родительский компонент `ArticleComponent` содержал порожденное представление: компонент `ArticleListComponent` или `ArticleDetailComponent`. Ради простоты в данном рецепте достаточно определить список статей в виде массива целочисленных значений.

Итак, определите следующую заготовку двух компонентов:

```
[app/article-list.component.ts]

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

@Component({
  template: `
    <h3>Article List</h3>
  `
})
export class ArticleListComponent {
  articleIds:Array<number> = [1,2,3,4,5];
}
[app/article-detail.component.ts]

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

@Component({
  template: `
    <h3>Article Detail</h3>
    <p>Showing article {{articleId}}</p>
  `
})
export class ArticleDetailComponent {
  articleId:number;
}
```

Определение порожденных маршрутов

В данный момент ничто не указывает на наличие в приложении порожденных маршрутов, и поэтому их необходимо теперь определить. Свойство `children` отдельного маршрута должно содержать очередной маршрут, который должен представлять вложенные маршруты, присоединяемые к родительскому маршруту.

Подобным образом определяется своего рода "дерево" маршрутизации, где каждый узел маршрута может иметь порожденные маршруты, определяемые рекурсивно. Подробнее об этом речь пойдет далее в главе.

Кроме того, следует воспользоваться обозначением параметров URL, чтобы объявить свойство `:articleId` как переменную в маршруте. Это даст возможность передать значения в маршруте, а затем извлечь эти значения из визуализируемого компонента.

Добавьте определения этих маршрутов следующим образом:

```
[app/app.module.ts]

import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {RouterModule, Routes} from '@angular/router';
import {RootComponent} from './root.component';
import {DefaultComponent} from './default.component';
import {ArticleComponent} from './article.component';
import {ArticleListComponent} from './article-list.component';
import {ArticleDetailComponent} from './article-detail.component';

const appRoutes:Routes = [
  {path: 'article', component: ArticleComponent,
    children: [
      {path: '', component: ArticleListComponent},
      {path: ':articleId', component: ArticleDetailComponent}
    ]
  },
  {path: '**', component: DefaultComponent},
];

@NgModule({
  imports: [
    BrowserModule,
    RouterModule.forRoot(appRoutes)
  ],
  declarations: [
    DefaultComponent,
    ArticleComponent,
    ArticleListComponent,
    ArticleDetailComponent,
    RootComponent
  ],
  bootstrap: [
    RootComponent
  ]
})
export class AppModule {}
```

Обратите внимание на то, что компонент ArticleListComponent обозначается ключом в виде пустой символьной строки. И в этом есть определенный смысл, поскольку каждый маршрут соединяется со своим родительским маршрутом, чтобы образовать полный маршрут. Если попытаться соединить каждый маршрут в этом дереве с его унаследованным маршрутом, чтобы получить полный путь, то сформированное только что определение маршрута будет состоять из следующих узлов дерева:

```
/ article => ArticleComponent
               ArticleListComponent
/ article/4 => ArticleComponent
                 ArticleDetailComponent<articleId=4>
/ **         => DefaultComponent
```

Количество конкретных маршрутов соответствует количеству узлов дерева URL, поскольку родительский маршрут статьи будет приводить к порожденному маршруту статьи. Хотя это равенство узлов дерева и маршрутов не всегда соблюдается, что зависит от конкретной конфигурации структуры маршрутов.

Определение ссылок на порожденные представления

Благодаря тому что маршруты приводят к порожденным компонентам, можно реализовать порожденные представления. Начав со списка типа ArticleList, создайте повторитель, чтобы сформировать ссылки на каждое из порожденных представлений, как показано ниже.

[app/article-list.component.ts]

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

@Component({
  template: `
    <h3>Article List</h3>
    <p *ngFor="let articleId of articleIds">
       <a [routerLink]="articleId">
         Article {{articleId}}
       </a>
    </p>
  `
})
export class ArticleListComponent {
  articleIds:Array<number> = [1,2,3,4,5];
}
```

Следует заметить, что директива **routerLink** привязывает к относительному пути, ведущему к подробному представлению. Текущим к данному представлению является путь **/article**, поэтому относительная ссылка **4** по директиве **routerLink** направит приложение по пути **/article/4** после щелчка кнопкой мыши.

Эти ссылки должны работать, но если щелкнуть на них кнопкой мыши, они приведут к подробному представлению, неспособному отобразить содержимое свойства :articleId из маршрута, поскольку оно еще не извлечено. Поэтому создайте в компоненте ArticleDetailComponent ссылку, которая направит пользователя обратно по маршруту article/, как показано ниже. В связи с тем что маршруты ведут себя как каталоги, можете просто воспользоваться относительным маршрутом, по которому пользователь перейдет на один уровень вверх по иерархии.

[app/article-detail.component.ts]

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

@Component({
  template: `
  <h3>Article Detail</h3>
   <p>Showing article {{articleId}}</p>
   <a [routerLink]="'../">Back up</a>
  `
})
export class ArticleDetailComponent {
  articleId:number;
}
```

Извлечение параметров маршрута

Главное отличие версий Angular 1 и 2 заключается в использовании конструкций наблюдаемых объектов типа Observable. В контексте маршрутизации наблюдаемые объекты применяются в Angular 2 с целью инкапсулировать маршрутизацию в виде последовательности событий и в конечном итоге сделать доступными значения, получаемые в разных состояниях этих событий.

Точнее говоря, параметры маршрута в Angular 2 доступны не напрямую, а через объект типа Observable в интерфейсе ActivatedRoute. Чтобы извлечь параметры маршрута, как только они станут доступны, можно установить наблюдатель в его наблюдаемом объекте params типа Observable. Итак, добавьте интерфейс ActivatedRoute и воспользуйтесь наблюдаемым объектом params типа Observable, чтобы извлечь свойство articleId и присвоить его члену экземпляра типа ArticleDetailComponent, как показано ниже. В итоге вы сможете заметить, что свойство articleId будет вставлено как параметр маршрута в компонент ArticleDetailComponent.

[app/article-detail/article-detail.component.ts]

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

```
@Component({
  template: `
    <h3>Article Detail</h3>
    <p>Showing article {{articleId}}</p>
    <a [routerLink]="'../'">Back up</a>
  `
})
export class ArticleDetailComponent {
  articleId:number;
  constructor(private activatedRoute_: ActivatedRoute) {
    activatedRoute_.params.subscribe(params =>
            this.articleId = params['articleId']);
  }
}
```

Принцип действия

В рассматриваемом здесь приложении применяются вложенные компоненты `AppComponent` и `ArticleComponent`, причем оба они содержат директиву `routerOutlet`. Каркас в состоянии принять определенную вами иерархию маршрутизации и привести ее к иерархии компонентов. В частности, для каждого маршрута, определенного в иерархии маршрутизации, должно быть равное количество директив `routerOutlet`, по которому они могут выполнить визуализацию.

Дополнение

Некоторым покажется странной потребность извлекать параметры маршрута из интерфейса наблюдаемых объектов типа `Observable`. Если такое решение покажется несколько громоздким, имеются способы привести его в порядок.

Рефакторинг с помощью асинхронных каналов

Напомним, что в Angular имеется возможность вставлять данные из наблюдаемых объектов непосредственно в шаблон, как только они будут готовы. Это особенно важно, поскольку следует ожидать, что наблюдаемый объект `params` типа `Observable` распространит данные лишь один раз. Следовательно, им можно воспользоваться, чтобы добавить свойство `articleId` в шаблон, не устанавливая наблюдатель явным образом, как показано ниже.

[app/article-detail.component.ts]

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

@Component({
  template: `
```

```
    <h3>Article Detail</h3>
    <p>Showing article
      {{(activatedRoute.params | async).articleId}}</p>
    <a [routerLink]="'../'">Back up</a>
  `
})
export class ArticleDetailComponent {
  constructor(activatedRoute: ActivatedRoute) {}
}
```

Несмотря на то что такой подход вполне работоспособен, применение закрытой ссылки на внедряемую службу в шаблоне может показаться несколько забавным. Более совершенная стратегия состоит в том, чтобы вставить в шаблон ссылку на требующийся открытый интерфейс наблюдаемых объектов типа `Observable` следующим образом:

```
[app/article-detail.component.ts]

import {Component} from '@angular/core';
import {Observable} from 'rxjs/Observable';
import {ActivatedRoute, Params} from '@angular/router';

@Component({
  template: `
    <h3>Article Detail</h3>
    <p>Showing article {{(params | async).articleId}}</p>
    <a [routerLink]="'../'">Back up</a>
  `
})
export class ArticleDetailComponent {
  params:Observable<Params>;
  constructor(private activatedRoute_: ActivatedRoute) {
    this.params = activatedRoute_.params;
  }
}
```

См. также

- Рецепт "Навигация по ссылкам с помощью директивы `routerLink`", в котором показано, каким образом осуществляется навигация в приложениях Angular.
- Рецепт "Навигация с помощью службы `Router`", где служба Angular применяется для навигации по приложению.
- Рецепт "Реализация сохраняющего состояние маршрута с помощью директивы `routerLinkActive`", в котором демонстрируется, как интегрировать поведение приложения с сохранением состояния URL.

- Рецепт "Обработка матричных параметров URL и массивов маршрутизации", в котором рассматривается поддержка в Angular матричных параметров URL.
- Рецепт "Добавление элементов управления аутентификацией маршрутов с помощью ограничителей маршрутов" с подробным описанием всего процесса настройки защищенных маршрутов в приложении.

Обработка матричных параметров URL и массивов маршрутизации

В каркасе Angular 2 внедрена собственная поддержка матричных параметров URL — замечательного средства, которое нередко упускается из виду. По существу, матричные параметры позволяют задавать произвольное количество данных в URL на *любом* уровне маршрутизации в Angular, а также читать эти данные как значение обычного параметра URL.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/4553/**.

Подготовка

Начните с версии приложения, полученной в конце раздела "Реализация" из представленного ранее рецепта "Реализация вложенных представлений с помощью параметров маршрута и порожденных маршрутов". Цель состоит в том, чтобы передать произвольные данные на уровнях объектов типа `ArticleList` и `ArticleDetail` в данном приложении только через URL.

Реализация

Массивы, задаваемые директивой `routerLink`, обрабатываются последовательно, и поэтому любая символьная строка, которая станет частью URL, будет содержать следующий далее объект, преобразованный в матричные параметры URL. Это проще уяснить на конкретном примере, поэтому начните с передачи каких-нибудь фиктивных параметров из директивы `routerLink` в представление типа `ArticleList`, как выделено ниже полужирным шрифтом.

```
[app/root.component.ts]

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

@Component({
  selector: 'root',
  template: `
    <h1>Root component</h1>
```

```
    <a [routerLink]="['']">
      Default</a>
    <a [routerLink]="['article', {listData: 'foo'}]">
      Article</a>
    <router-outlet></router-outlet>
  `
})
export class RootComponent {}
```

Если теперь щелкнуть на данной ссылке, браузер перейдет по следующему пути, успешно визуализируя представление типа `ArticleList`:

```
/article;listData=foo
```

Для доступа к значениям параметров их достаточно извлечь из объекта типа `ActivatedRoute` следующим образом:

```
[app/article-list.component.ts]

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

@Component({
  template: `
    <h3>Article List</h3>
    <p *ngFor="let articleId of articleIds">
      <a [routerLink]="[articleId]">
        Article {{articleId}}
      </a>
    </p>
  `
})
export class ArticleListComponent {
  articleIds:Array<number> = [1,2,3,4,5];
  constructor(private activatedRoute_:ActivatedRoute) {
    activatedRoute_.params
      .subscribe(params => {
        console.log('List params:');
        console.log(window.location.href)
        console.log(params);
      });
  }
}
```

Как только загрузится представление, появятся следующие данные:

```
List params:
/article;listData=foo
Object {listData: "foo"}
```

Замечательно! А теперь сделайте то же самое с подробным представлением, как выделено ниже полужирным шрифтом.

```
[app/article-list.component.ts]

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

@Component({
  template: `
    <h3>Article List</h3>
    <p *ngFor="let articleId of articleIds">
      <a [routerLink]="[articleId, {detailData: 'bar'}]">
        Article {{articleId}}
      </a>
    </p>
  `
})
export class ArticleListComponent {
  articleIds:Array<number> = [1,2,3,4,5];
  constructor(private activatedRoute_:ActivatedRoute) {
    activatedRoute_.params
      .subscribe(params => {
        console.log('List params:');
        console.log(window.location.href)
        console.log(params);
      });
  }
}
```

Добавьте в подробное представление тот же самый диагностический вывод на консоль, как показано ниже.

```
[app/article-detail.component.ts]

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

@Component({
  template: `
    <h3>Article Detail</h3>
    <p>Showing article {{articleId}}</p>
    <a [routerLink]="'../'">Back up</a>
  `
})
export class ArticleDetailComponent {
  articleId:number;
  constructor(private activatedRoute_:ActivatedRoute) {
    activatedRoute_.params
      .subscribe(params => {
        console.log('Detail params:');
```

```
        console.log(window.location.href)
        console.log(params);
        this.articleId = params['articleId']
      });
  }
}
```

Посетив страницу с подробным представлением, вы обнаружите следующую выведенную диагностическую информацию:

```
Detail params:
/article;listData=foo/1;detailData=bar
Object {articleId: "1", detailData: "foo"}
```

Любопытно, что Angular в состоянии не только связывать разные матричные параметры с различными уровнями маршрутизации, но и объединять ожидаемый параметр `articleId` с неожидаемым параметром `detailData` в одну и ту же порцию данных, распространяемых наблюдаемым объектом.

Принцип действия

Каркас Angular способен плавно выполнить преобразование массива маршрутизации, содержащего объект матричных параметров, в сериализированный URL, содержащий матричные параметры, а затем обратно в десериализированный объект JavaScript, содержащий значения параметров. Это дает возможность хранить произвольные данные в URL на разных уровнях, не втискивая их в символьную строку запроса, присоединяемую в конце URL.

Дополнение

Обратите внимание на то, что если щелкнуть на ссылке `Back up` (Резервировать) в подробном представлении, то сохранится параметр `listData`, присутствующий в URL. Каркас Angular будет добросовестно поддерживать состояние при перемещении по приложению, поэтому матричные параметры позволяют весьма эффективно хранить данные, фиксирующие состояние, при перемещении или загрузке страниц.

См. также

- Рецепт "Навигация по ссылкам с помощью директивы `routerLink`", в котором демонстрируется, каким образом осуществляется навигация по приложениям Angular.
- Рецепт "Навигация с помощью службы `Router`", где служба Angular применяется для навигации по приложению.

- Рецепт "Реализация сохраняющего состояние маршрута с помощью директивы routerLinkActive", в котором показано, как интегрировать поведение приложения с сохранением состояния URL.
- Рецепт "Реализация вложенных представлений с помощью параметров маршрута и порожденных маршрутов" с примером, демонстрирующим порядок настройки URL в Angular для поддержки вложения и передачи данных.

Добавление элементов управления аутентификацией маршрутов с помощью ограничителей маршрутов

Характер одностраничных приложений, полностью управляющих процессом маршрутизации, позволяет управлять каждой стадией данного процесса. Это означает, что изменения, происходящие в маршруте, можно перехватывать и принимать решения, куда следует направить пользователя.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/6135/**.

Подготовка

В данном рецепте предстоит разработать простое псевдоаутентификационное приложение. Цель состоит в том, чтобы скрыть от пользователей определенные представления, когда они не проходят аутентификацию, и в то же время реализовать правильную последовательность входа и выхода из системы.

Реализация

Начните с определения двух первоначальных представлений в своем приложении. Одним из них должно быть стандартное представление, доступное каждому, а другим — профильное представление, которое будет доступно для просмотра аутентифицированным пользователям. Очевидно, что приведенный ниже прикладной код пока еще ничего не делает.

[app/app.module.ts]

```
import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {RouterModule, Routes} from '@angular/router';
import {RootComponent} from './root.component';
import {DefaultComponent} from './default.component';
```

```
import {ProfileComponent} from './profile.component';

const appRoutes:Routes = [
  {path: 'profile', component: ProfileComponent},
  {path: '**', component: DefaultComponent}
];

@NgModule({
  imports: [
    BrowserModule,
    RouterModule.forRoot(appRoutes)
  ],
  declarations: [
    DefaultComponent,
    ProfileComponent,
    RootComponent
  ],
  bootstrap: [
   RootComponent
  ]
})
export class AppModule {}

[app/default.component.ts]

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

@Component({
  template: `
    <h2>Default view!</h2>
  `
})
export class DefaultComponent {}

[app/profile.component.ts]

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

@Component({
  template: `
    <h2>Profile view</h2>
      Username: <input>
    <button>Update</button>
  `
})
export class ProfileComponent {}
```

Реализация службы аутентификации

Как было показано в главе 5, требуется реализовать службу, которая будет полностью поддерживать состояние в объекте типа `BehaviorSubject`.

Напомним, что объект типа **BehaviorSubject** будет распространять свое последнее отправляемое значение всякий раз, когда на него подпишется наблюдатель. И для этого требуется установить исходное состояние. Впрочем, для службы аутентификации сделать это совсем не трудно, начав ее работу с неаутентифицированного состояния.

Для целей данного рецепта допустим, что пустое (null) имя пользователя означает, что он не аутентифицирован. А любое другое строковое значение означает, что пользователь прошел аутентификацию.

```
[app/auth.service.ts]

import {Injectable} from '@angular/core';
import {BehaviorSubject} from 'rxjs/BehaviorSubject';
import {Observable} from 'rxjs/Observable';

@Injectable()
export class AuthService {
  private authSubject_:BehaviorSubject<any> =
          new BehaviorSubject(null);
  usernameEmitter:Observable<string>;

  constructor() {
    this.usernameEmitter = this.authSubject_.asObservable();
    this.logout();
  }

  login(username:string):void {
    this.setAuthState_(username);
  }

  logout():void {
    this.setAuthState_(null);
  }

  private setAuthState_(username:string):void {
    this.authSubject_.next(username);
  }
}
```

Следует иметь в виду, что имя пользователя нигде не сохраняется как символьная строка. Состояние аутентификации полностью хранится в объекте типа BehaviorSubject.

Подключение к профильному представлению

Сделайте рассматриваемую здесь службу доступной во всем приложении и подключите профильное представление, как показано ниже.

[app/app.module.ts]

```
import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {RouterModule, Routes} from '@angular/router';
import {RootComponent} from './root.component';
import {DefaultComponent} from './default.component';
import {ProfileComponent} from './profile.component';
import {AuthService} from './auth.service';

const appRoutes:Routes = [
  {path: 'profile', component: ProfileComponent},
  {path: '**', component: DefaultComponent}
];

@NgModule({
  imports: [
    BrowserModule,
    RouterModule.forRoot(appRoutes)
  ],
  declarations: [
    DefaultComponent,
    ProfileComponent,
    RootComponent
  ],
  providers: [
    AuthService
  ],
  bootstrap: [
    RootComponent
  ]
})
export class AppModule {}
```

[app/profile.component.ts]

```
import {Component} from '@angular/core';
import {AuthService} from './auth.service';
import {Observable} from 'rxjs/Observable';

@Component({
  template: `
    <h2>Profile view</h2>
    Username: <input #un value="{{username | async}}">
    <button (click)=update(un.value)>Update</button>
  `
})
export class ProfileComponent {
  username:Observable<string>;
```

```
  constructor(private authService_:AuthService) {
    this.username = authService_.usernameEmitter;
  }

  update(username:string):void {
    this.authService_.login(username);
  }
}
```

При вставке значений очень удобно пользоваться каналом **async**. Напомним, что, вызывая метод **subscribe()** для наблюдаемого объекта данной службы из полученного экземпляра компонента представления, необходимо также вызвать метод **unsubscribe()** для объекта типа **Subscription**, когда компонент удаляется. Иначе в приложении произойдет утечка приемника событий. Этого можно избежать, сделав наблюдаемый объект доступным для представления.

Подключив профильное представление, добавьте ссылки и вставьте имя пользователя в корневое представление приложения на панели навигации, чтобы получить возможность для навигации. Для этого достаточно добавить все ссылки, требующиеся в данном рецепте, следующим образом:

```
[app/root.component.ts]

import {Component} from '@angular/core';
import {Router} from '@angular/router';
import {AuthService} from './auth.service';
import {Observable} from 'rxjs/Observable';

@Component({
  selector: 'root',
  template: `
    <h3 *ngIf="!!(username | async)">
      Hello, {{username | async}}.
    </h3>
    <a [routerLink]="['']">Default</a>
    <a [routerLink]="['profile']">Profile</a>
    <a *ngIf="!!(username | async)"
      [routerLink]="['login']">Login</a>
    <a *ngIf="!!(username | async)"
      [routerLink]="['logout']">Logout</a>
    <router-outlet></router-outlet>
  `
})
export class RootComponent {
  username:Observable<string>;

  constructor(private authService_:AuthService) {
    this.username = authService_.usernameEmitter;
  }
}
```

Ради совместимости и с целью упростить определение компонента в данном случае применяется канал **async**. Но поскольку в шаблоне имеются четыре экземпляра, ссылающиеся на один и тот же наблюдаемый объект, то не лучше ли установить по ходу дела один подписчик на наблюдаемый объект, привязать его к строковому члену в объекте типа **RootComponent** и интерполировать полученный результат. Этого нетрудно добиться благодаря поддерживаемой в Angular привязке данных, но подписчик все равно придется снять с регистрации при его удалении. А поскольку речь идет о корневом компоненте приложения, то не следует ожидать, что это произойдет автоматически.

Ограничение доступа к маршруту

Все бы ничего, но обратите внимание, что профильное представление в любом случае разрешает пользователю входить в систему. Поэтому необходимо каким-то образом ограничить доступ к данному представлению, разрешив пользователю обращаться к нему только в том случае, если он аутентифицирован. Ограничитель маршрутов, доступный в Angular, дает возможность выполнить прикладной код, проверить маршрут и выполнить переадресацию по мере надобности, прежде чем произойдет навигация.

Термин *ограничитель маршрутов* (route guard) может несколько озадачить. Это средство следует рассматривать как прокладку для маршрута, где можно добавить логику, которая выполняется в Angular непосредственно перед следованием по новому маршруту. Ограничитель может действительно защищать маршрут от неаутентифицированного пользователя. Но в то же время он может легко переадресовывать по условию, сохранять текущий URL или выполнять другие операции. А поскольку ограничитель маршрутов должен быть оснащен декоратором **@Injectable**, то его лучше рассматривать как тип службы.

Начните со следующей заготовки службы типа `AuthGuardService`, определенной в новом файле для ограничителей маршрутов:

```
[app/route-guards.service.ts]

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

@Injectable()
export class AuthGuardService implements CanActivate {
  constructor() {}

  canActivate() {
    // Этот метод вызывается во время изменений в маршруте,
    // если данный класс перечислен в массиве типа Routes
  }
}
```

Прежде чем служба типа `AuthGuardService` вступит в действие, импортируйте и добавьте ее модуль в массив типа `Routes`, как выделено ниже полужирным шрифтом.

```
[app/app.module.ts]

import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {RouterModule, Routes} from '@angular/router';
import {RootComponent} from './root.component';
import {DefaultComponent} from './default.component';
import {ProfileComponent} from './profile.component';
import {AuthService} from './auth.service';
import {AuthGuardService} from './route-guards.service';

const appRoutes:Routes = [
  {
    path: 'profile',
    component: ProfileComponent,
    canActivate: [AuthGuardService]
  },
  {
    path: '**',
    component: DefaultComponent
  }
];

@NgModule({
  imports: [
    BrowserModule,
    RouterModule.forRoot(appRoutes)
  ],
  declarations: [
    DefaultComponent,
    ProfileComponent,
    RootComponent
  ],
  providers: [
    AuthService,
    AuthGuardService
  ],
  bootstrap: [
    RootComponent
  ]
})
export class AppModule {}
```

Теперь метод `canActivate()`, определенный в службе типа `AuthGuard Service`, вызывается всякий раз, когда в приложении обнаруживается совпадение с маршрутом и предпринимается попытка последовать по данному

маршруту. Истинное значение, возвращаемое данным методом, означает, что навигация может произойти. А ложное возвращаемое значение означает, что навигация отменена.

Метод **canActivate()** может возвратить логическое значение типа **boolean** или **Observable<boolean>**. Однако, если возвратить логическое значение типа **Observable<boolean>**, приложение добросовестно подождет до тех пор, пока наблюдаемый объект не распространит свое значение, и только после этого совершит навигацию.

В приложении состояние аутентификации находится в объекте типа BehaviorSubject, и поэтому данному методу нужно лишь подписаться, проверить имя пользователя и совершить навигацию, если это имя не пустое. Оно соответственно обрабатывается в данном методе, чтобы возвратить значение типа Observable<boolean>, как показано ниже.

```
[app/route-guards.service.ts]

import {Injectable} from '@angular/core';
import {CanActivate, Router} from '@angular/router';
import {AuthService} from './auth.service';
import {Observable} from 'rxjs/Observable';

@Injectable()
export class AuthGuardService implements CanActivate {
  constructor(private authService_:AuthService,
              private router_:Router) {}

  canActivate():Observable<boolean> {
    return this.authService_.usernameEmitter.map(
       username => {
         if (!username) {
           this.router_.navigate(['login']);
         } else {
           return true;
         }
    });
  }
}
```

Реализовав это, вы заметите, что навигация вообще не произойдет, несмотря на то, что служба правильно распространяет имя пользователя. Дело в том, что получатель значения, возвращаемого методом canActivate(), ожидает не распространения данных от наблюдаемого объекта, а *завершения* этого процесса. В данном случае требуется лишь выявить имя пользователя в объекте типа BehaviorSubject, и поэтому можно возвратить лишь новый

наблюдаемый объект, который, в свою очередь, возвратит одно значение и завершит процесс распространения с помощью метода `take()`, как выделено ниже полужирным шрифтом.

```
[app/route-guards.service.ts]

import {Injectable} from '@angular/core';
import {CanActivate, Router} from '@angular/router';
import {AuthService} from './auth.service';
import {Observable} from 'rxjs/Observable';
import 'rxjs/add/operator/take';

@Injectable()
export class AuthGuardService implements CanActivate {
  constructor(private authService_:AuthService,
  private router_:Router) {}

  canActivate():Observable<Boolean> {
    return this.authService_.usernameEmitter.map(
        username => {
          if (!username) {
            this.router_.navigate(['login']);
          } else {
            return true;
          }
    }).take(1);
  }
}
```

Замечательно! Тем не менее данному приложению все еще недостает метода для формального входа и выхода из системы.

Добавление поведения для входа в систему

Странице входа в систему потребуется свое представление, и поэтому она должна получить свой маршрут и компонент. Как только пользователь войдет в систему, удерживать его на странице входа в систему больше нет нужды, а следовательно, пользователя требуется переадресовать к стандартному представлению по окончании регистрации. Прежде всего создайте компонент регистрации и соответствующее его представление следующим образом:

```
[app/login.component.ts]

import {Component} from '@angular/core';
import {Router} from '@angular/router';
import {AuthService} from './auth.service';

@Component({
  template: `
    <h2>Login view</h2>
```

```
    <input #un>
    <button (click)="login(un.value)">Login</button>
  `
})
export class LoginComponent {
  constructor(private authService_:AuthService,
  private router_:Router) { }
  login(newUsername:string):void {
    this.authService_.login(newUsername);
    this.authService_.usernameEmitter
        .subscribe(username => {
           if (!!username) {
             this.router_.navigate(['']);
           }
        });
  }
}
```

[app/app.module.ts]

```
import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {RouterModule, Routes} from '@angular/router';
import {RootComponent} from './root.component';
import {DefaultComponent} from './default.component';
import {ProfileComponent} from './profile.component';
import {LoginComponent} from './login.component';
import {AuthService} from './auth.service';
import {AuthGuardService} from './route-guards.service';

const appRoutes:Routes = [
  {
    path: 'login',
    component: LoginComponent
  },
  {
    path: 'profile',
    component: ProfileComponent,
    canActivate: [AuthGuardService]
  },
  {
    path: '**',
    component: DefaultComponent
  }
];

@NgModule({
  imports: [
    BrowserModule,
    RouterModule.forRoot(appRoutes)
  ],
  declarations: [
```

```
    LoginComponent,
    DefaultComponent,
    ProfileComponent,
    RootComponent
  ],
  providers: [
    AuthService,
    AuthGuardService
  ],
  bootstrap: [
    RootComponent
  ]
})
export class AppModule {}
```

Теперь пользователь сможет войти в систему. Все это, конечно, замечательно, но обратите внимание на то, что при такой реализации обновление имени пользователя в представлении профиля приведет к выводу стандартного представления, проявляя такое же самое поведение, которое определено в компоненте регистрации. Дело в том, что подписчик по-прежнему прослушивает наблюдаемый объект типа `AuthService`. Поэтому нам придется добавить метод `OnDestroy()`, чтобы правильно удалить представление регистрации, как показано ниже.

```
[app/login.component.ts]

import {Component, ngOnDestroy} from '@angular/core';
import {Router} from '@angular/router';
import {AuthService} from './auth.service';
import {Subscription} from 'rxjs/Subscription';

@Component({
  template: `
    <h2>Login view</h2>
    <input #un>
    <button (click)="login(un.value)">Login</button>
  `
})
export class LoginComponent implements OnDestroy {
  private usernameSubscription_:Subscription;
  constructor(private authService_:AuthService,
    private router_:Router) { }

  login(newUsername:string):void {
    this.authService_.login(newUsername);
    this.usernameSubscription_ = this.authService_
        .usernameEmitter
        .subscribe(username => {
          if (!!username) {
```

```
            this.router_.navigate(['']);
          }
        });
  }

  ngOnDestroy() {
    // вызвать метод unsubscribe() только в том случае,
    // если данное представление существует
    this.usernameSubscription_ &&
    this.usernameSubscription_.unsubscribe();
  }
}
```

Добавление поведения для выхода из системы

И, наконец, требуется предоставить пользователю возможность каким-то образом выйти из системы. Этого можно добиться самыми разными способами, но наиболее удачная реализация состоит в том, чтобы поручить выход из системы соответствующим методам, не внедряя слишком много стереотипного кода.

В идеальном случае приложение должно быть в состоянии просто перейти по маршруту выхода из системы, предоставив Angular возможность сделать все остальное автоматически. И этого можно добиться с помощью метода `canActivate()`. Прежде всего определите новый ограничитель маршрутов, как показано ниже. Такое поведение не требует особых пояснений.

[app/route-guards.service.ts]

```
import {Injectable} from '@angular/core';
import {CanActivate, Router} from '@angular/router';
import {AuthService} from './auth.service';
import {Observable} from 'rxjs/Observable';
import 'rxjs/add/operator/take';

@Injectable()
export class AuthGuardService implements CanActivate {
  constructor(private authService_:AuthService,
    private router_:Router) {}

  canActivate():Observable<boolean> {
    return this.authService_.usernameEmitter.map(
        username => {
          if (!username) {
            this.router_.navigate(['login']);
          } else {
            return true;
          }
    }).take(1);
  }
}
```

```
@Injectable()
export class LogoutGuardService implements CanActivate {
  constructor(private authService_:AuthService,
              private router_:Router) {}

  canActivate():boolean {
    this.authService_.logout();
    this.router_.navigate(['']);
    return true;
  }
}
```

Метод **canActivate()** должен иметь такую же сигнатуру, которая определена в интерфейсе **CanActivate**. И хотя это всегда будет приводить к новому представлению, следует добавить возвращаемое значение, чтобы удовлетворить компилятор и учесть любые случаи, когда работа приведенного выше кода завершится аварийно.

Далее добавьте компонент выхода из системы и маршрут. Компонент выхода из системы вообще не подлежит визуализации, но определение маршрута требуется сопоставить с корректным компонентом. Таким образом, компонент выхода из системы `LogoutComponent` будет состоять из фиктивного класса, как выделено ниже полужирным шрифтом. В итоге процесс входа и выхода из системы окажется полностью работоспособным.

```
[app/logout.component.ts}

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

@Component({
  template: ''
})

export class LogoutComponent{}

[app/app.module.ts]

import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {RouterModule, Routes} from '@angular/router';
import {RootComponent} from './root.component';
import {DefaultComponent} from './default.component';
import {ProfileComponent} from './profile.component';
import {LoginComponent} from './login.component';
import {LogoutComponent} from './logout.component';
import {AuthService} from './auth.service';
import {AuthGuardService, LogoutGuardService}
        from './route-guards.service';
```

```
const appRoutes:Routes = [
  {
    path: 'login',
    component: LoginComponent
  },
  {
    path: 'logout',
    component: LogoutComponent,
    canActivate: [LogoutGuardService]
  },
  {
    path: 'profile',
    component: ProfileComponent,
    canActivate: [AuthGuardService]
  },
  {
    path: '**',
    component: DefaultComponent
  }
];

@NgModule({
  imports: [
    BrowserModule,
    RouterModule.forRoot(appRoutes)
  ],
  declarations: [
    LoginComponent,
    LogoutComponent,
    DefaultComponent,
    ProfileComponent,
    RootComponent
  ],
  providers: [
    AuthService,
    AuthGuardService,
    LogoutGuardService
  ],
  bootstrap: [
    RootComponent
  ]
})
export class AppModule {}
```

Принцип действия

Рассматриваемая здесь реализация построена на основе наблюдаемых объектов и ограничителей маршрутов. Наблюдаемые объекты позволяют хранить состояние в модуле `AuthService`, делая его доступным одновре-

менно через объект типа `BehaviorSubject`, а ограничители маршрутов — перемещаться по условию и производить переадресацию на усмотрение приложения.

Дополнение

Безопасность приложения — обширный и сложный предмет. В рассматриваемом здесь рецепте демонстрируется, насколько плавно пользователь может перемещаться по приложению, хотя такую модель его безопасности ни в коей мере нельзя считать строгой.

Реальная аутентификация

Следует всегда допускать, что клиент может манипулировать своей средой выполнения. Даже если защитить как можно тщательнее в данном примере методы входа и выхода из системы, определенные в модуле `AuthService`, пользователю не составит особого труда получить доступ к этим методам и пройти аутентификацию.

Безопасность пользовательских интерфейсов, к которым имеют непосредственное отношение приложения Angular, изначально не предполагается. В модели "клиент–сервер" обязанности по соблюдению безопасности возлагаются на сервер, поскольку пользователь не контролирует эту среду выполнения. В конкретном приложении метод `login()` сделает сетевой запрос к серверу на получение маркера (token). В двух наиболее распространенных реализациях аутентификации с помощью маркеров по стандарту JSON Web Tokens и cookie-файлов это делается по-разному, хотя они, по существу, являются лишь вариациями на одну и ту же тему. Каркас Angular и браузер будут хранить и посылать подобные маркеры, но в конечном счете сервер должен действовать как шлюз для доступа к защищенной информации.

Защищенные данные и представления

Любая защищенная информация, которую можно послать клиенту, должна пройти аутентификацию на сервере. Для многих разработчиков это очевидный факт, особенно в том случае, если приходится обращаться к интерфейсу API. Но каркас Angular запрашивает также у сервера шаблоны и статические файлы, причем некоторые из них могут и не потребоваться для обслуживания "левых" пользователей. В данном случае придется сконфигурировать сервер для аутентификации запросов на эти статические файлы, прежде чем они будут предоставлены клиенту.

См. также

- Рецепт "Навигация с помощью службы `Router`", в котором служба Angular применяется для навигации по приложению.

- Рецепт "Реализация сохраняющего состояние маршрута с помощью директивы `routerLinkActive`", где демонстрируется, как интегрировать поведение приложения с сохранением состояния URL.
- Рецепт "Реализация вложенных представлений с помощью параметров маршрута и порожденных маршрутов" с примером, демонстрирующим порядок настройки URL в Angular для поддержки вложения и передачи данных.
- Рецепт "Обработка матричных параметров URL и массивов маршрутизации", в котором рассматривается поддержка в Angular матричных параметров URL.

Глава 7

Службы, внедрение зависимостей и класс NgModule

В этой главе даются следующие рецепты.

- Внедрение простой службы в компонент
- Контроль над созданием экземпляра службы и внедрением средствами класса `NgModule`
- Назначение псевдонимов при внедрении служб с помощью свойств `useClass` и `useExisting`
- Внедрение значения в виде службы с помощью свойства `useValue` и класса `OpaqueToken`
- Создание настраиваемой поставщиком службы с помощью свойства `useFactory`

Введение

В версии Angular 1 предоставлялись самые разные виды служб. Многие из них по большей части перекрывали своими возможностями друг друга и были неудобны. И все они были одиночками.

Такой принцип создания служб был полностью отвергнут в версии Angular 2. Вместо этого появилась совершенно новая система внедрения зависимостей, которая оказалась намного более удобной и расширяемой, чем ее предшественница. Она дает в распоряжение разработчиков как атомарные, так и неатомарные типы служб, фабрики, средства совмещения имен и самые разные чрезвычайно полезные инструментальные средства для применения в их приложениях.

Если вам требуется применять службы таким же образом, как и прежде, вы сможете легко перенести свое представление о типах служб и в новую систему. Но тем разработчикам, которым требуется извлечь наибольшую пользу из своих приложений, новая система внедрения зависимостей принесет немалые выгоды в написании масштабируемых приложений.

Внедрение простой службы в компонент

Самым типичным примером может служить внедрение службы в сам компонент. Несмотря на то что порядок определения типов служб и внедрения зависимостей остается в основном прежним, очень важно иметь ясное представление об основах системы внедрения зависимостей в Angular 2, поскольку она имеет ряд заметных отличий.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/4263/**.

Подготовка

Допустим, имеется приведенная ниже заготовка приложения. Цель состоит в том, чтобы реализовать службу, которую можно внедрить в данный компонент и возвратить название статьи, заполнив шаблон.

```
[app/root.component.ts]

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

@Component({
  selector: 'root',
  template: `
    <h1>root component!</h1>
    <button (click)="fillArticle()">Show article</button>
    <h2>{{title}}</h2>
  `
})
export class RootComponent {
  title:string;
  constructor() {}
  fillArticle() {}
}
```

Реализация

Как и следовало ожидать, службы в Angular 2 представлены в виде классов. Аналогично компонентам, службы обозначаются декоратором `@Injectable`. Итак, создайте службу в отдельном файле следующим образом:

```
[app/article.service.ts]

import {Injectable} from '@angular/core';

@Injectable()
export class ArticleService {
  private title_:string = `
    CFO Yodels Quarterly Earnings Call, Stock Skyrockets
  `
}
```

У данной службы имеется закрытое название, которое требуется перенести в компонент, но прежде саму службу необходимо сделать доступной для компонента. Для этого достаточно импортировать службу, а затем указать ее в свойстве `providers` из модуля приложения, как выделено ниже полужирным шрифтом.

```
[app/app.module.ts]

import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {RootComponent} from './root.component';
import {ArticleService} from './article.service';

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

А теперь, когда можно предоставить службу, внедрите ее в компонент следующим образом:

```
[app/root.component.ts]

import {Component} from '@angular/core';
import {ArticleService} from './article.service';

@Component({
  selector: 'root',
```

```
  template: `
    <h1>root component!</h1>
    <button (click)="fillArticle()">Show article</button>
    <h2>{{title}}</h2>
  `
})
export class RootComponent {
  title:string;

  constructor(private articleService_:ArticleService) {}

  fillArticle() {}
}
```

В приведенном выше коде создается новый экземпляр службы типа `ArticleService` при получении экземпляра типа `RootComponent`, а затем данная служба внедряется в конструктор. Все, что внедряется в компонент, будет доступно в виде его члена, которым можно воспользоваться для подключения метода службы к методу компонента, как показано ниже.

```
[app/article.service.ts]

import {Injectable} from '@angular/core';

@Injectable()
export class ArticleService {
  private title_:string = `
    CFO Yodels Quarterly Earnings Call, Stock Skyrockets
  `;
  getTitle() {
    return this.title_;
  }
}
[app/root.component.ts]

import {Component} from '@angular/core';
import {ArticleService} from './article.service';

@Component({
  selector: 'root',
  template: `
    <h1>root component!</h1>
    <button (click)="fillArticle()">Show article</button>
    <h2>{{title}}</h2>
  `
})
export class RootComponent {
  title:string;

  constructor(private articleService_:ArticleService) {}
```

```
  fillArticle():void {
    this.title = this.articleService_.getTitle();
  }
}
```

Принцип действия

В отсутствие декоратора созданная только что служба довольно проста по своей композиции. Благодаря декорации `@Injectable()` класс обозначается в Angular как доступный для внедрения в любом другом месте.

В связи с обозначением компонентов как внедряемых возникает целый ряд вопросов, которые следует иметь в виду, чтобы отличать их от компонентов, передаваемых в качестве параметров. В частности, когда получается экземпляр внедряемого класса? Как он связывается с экземпляром компонента? Каким образом контролируются глобальные и локальные экземпляры? Все подобные вопросы обсуждаются более подробно в рецептах, следующих далее в этой главе.

Обозначение службы как внедряемой — это лишь часть общей картины. Компонент еще нужно известить о существовании службы. Сначала следует импортировать класс службы в модуль компонента, но одного этого явно недостаточно. Напомним, что синтаксис, применяемый для внедрения службы, просто служит для того, чтобы указать ее в качестве параметра конструктора. А внутренний механизм Angular достаточно развит логически, чтобы распознавать эти параметры и аргументы как внедряемые компоненты. Но для этого требуется завершающая общую картину часть, чтобы соединить импортированный модуль с тем местом, где он будет служить в качестве внедряемого ресурса.

Эта завершающая часть принимает форму свойства `providers` в определении класса `NgModule`. Для целей данного рецепта совсем не важно знать особенности этого свойства. Но если кратко, то оно представляет массив, где параметр конструктора `articleService` обозначается как внедряемый, а служба типа `ArticleService` должна быть внедрена в конструктор.

Дополнение

Следует признать, что в данном рецепте декораторы языка TypeScript помогают устанавливать внедрение зависимостей. Декораторы модифицируют не экземпляр, а скорее определение класса. Экземпляр типа `NgModule`, где содержится список поставщиков, будет инициализирован прежде любого другого получаемого экземпляра конкретного компонента. Таким образом, каркасу Angular станут известны все службы, которые, возможно, придется внедрить в конструктор.

См. также

- Рецепт "Контроль над созданием экземпляра службы и внедрением средствами класса NgModule", в котором дается обширный обзор, каким образом в Angular 2 строятся иерархии поставщиков с помощью модулей.

Контроль над созданием экземпляра службы и внедрением средствами класса NgModule

Версия Angular 2 заметно отличается от версии Angular 1.x наличием иерархической системы внедрения зависимостей. Из этого обстоятельства вытекает немало существенных последствий, и к самым примечательным из них относится способность контролировать, когда и сколько служб создается.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/2102/**.

Подготовка

Допустим, имеется следующая заготовка приложения:

```
[app/root.component.ts]

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

@Component({
  selector: 'root',
  template: `
    <h1>root component!</h1>
    <article></article>
    <article></article>
  `
})
export class RootComponent {}

[app/article.component.ts]

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

@Component({
  selector: 'article',
  template: `
    <p>Article component!</p>
  `
})
export class ArticleComponent {}

[app/article.service.ts]
```

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

@Injectable()
export class ArticleService {
  constructor() {
    console.log('ArticleService constructor!');
  }
}
```

```
[app/app.module.ts]

import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {RootComponent} from './root.component';
import {ArticleComponent} from './article.component;

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

Цель состоит в том, чтобы внедрить единственный экземпляр службы типа `ArticleService` в два порожденных компонента. В данном рецепте метод `console.log()`, вызываемый в конструкторе класса `ArticleService`, позволяет выяснить, когда именно получен отдельный экземпляр службы данного типа.

Реализация

Начните с импорта службы в класс `AppModule`, а затем предоставьте ее следующим образом:

```
[app/app.module.ts]

import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {RootComponent} from './root.component';
import {ArticleComponent} from './article.component;
import {ArticleService} from './article.service;

@NgModule({
```

```
  imports: [
    BrowserModule,
  ],
  declarations: [
    RootComponent,
    ArticleComponent
  ],
  providers: [
    ArticleService
  ]
  bootstrap: [
    RootComponent
  ]
})
export class AppModule {}
```

Служба типа `ArticleService` предоставляется в том же самом модуле, где объявлен компонент `ArticleComponent`, поэтому службу типа `ArticleService` можно внедрить теперь в порожденные экземпляры типа `ArticleComponent`, как показано ниже. Таким образом, один и тот же экземпляр службы внедряется в порожденные компоненты, поскольку метод `console.log()` выполняется в конструкторе класса `ArticleService` лишь один раз.

[app/article/article.component.ts]

```
import {Component} from '@angular/core';
import {ArticleService} from './article.service';

@Component({
  selector: 'article',
  template: `
    <p>Article component!</p>
  `
})
export class ArticleComponent {
  constructor(private articleService_:ArticleService) {}
}
```

Разделение корневого модуля

По мере расширения приложения остается все меньше оснований втискивать все подряд в один и тот же модуль верхнего уровня. Вместо этого было бы идеально разбить модули на отдельные значимые части. А в данном рецепте было бы предпочтительно предоставить службу типа `ArticleService` тем частям приложения, где их требуется внедрить.

Определите далее новый исходный файл для модуля ArticleModule и перенесите в него операторы импорта соответствующих модулей, как показано ниже.

```
[app/article.module.ts]

import {NgModule} from '@angular/core';
import {ArticleComponent} from './article.component';
import {ArticleService} from './article.service';

@NgModule({
  declarations: [
    ArticleComponent
  ],
  providers: [
    ArticleService
  ],
  bootstrap: [
    ArticleComponent
  ]
})
export class ArticleModule {}
```

Затем импортируйте весь этот модуль в модуль AppModule следующим образом:

```
[app/app.module.ts]

import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {RootComponent} from './root.component';
import {ArticleModule} from './article.module';

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

Остановившись на этом, вы не обнаружите в данном коде ошибок, тем не менее, модуль AppModule не в состоянии визуализировать компонент ArticleComponent. Дело в том, что в модулях Angular, как и в других

модульных системах, необходимо явно определить, что именно экспортируется в другие модули. Именно это и выделено ниже полужирным шрифтом. Таким образом, экземпляр службы типа ArticleService получается лишь один раз.

```
[app/article.module.ts]

import {NgModule} from '@angular/core';
import {ArticleComponent} from './article.component';
import {ArticleService} from './article.service';

@NgModule({
  declarations: [
    ArticleComponent
  ],
  providers: [
    ArticleService
  ],
  bootstrap: [
   ArticleComponent
  ],
  exports: [
    ArticleComponent
  ]
})
export class ArticleModule {}
```

Принцип действия

Иерархическая структура системы внедрения зависимостей в Angular 2 выгодно используется для предоставления и внедрения служб. Там, где служба внедрена, средствами Angular будет получен экземпляр данной службы, где бы она ни предоставлялась. В определении модуля это происходит лишь один раз.

В данном случае служба типа ArticleService предоставляется обоим модулям: AppModule и ArticleModule. Несмотря на то что данная служба внедряется дважды (по одному разу в каждом экземпляре типа ArticleComponent), объявление свойства providers в Angular используется с целью решить, когда именно следует создавать службу.

Дополнение

В данный момент у любознательного разработчика может возникнуть немало вопросов, касающихся поведения системы внедрения зависимостей. Имеются многочисленные разновидности конфигурации, которые могут

оказаться полезными для разработчиков, и эти конфигурации требуют лишь минимальной коррекции кода из предыдущего результата.

Внедрение других экземпляров служб в разные компоненты

Как следует из предыдущего пояснения, рассматриваемое здесь приложение можно переконфигурировать, чтобы внедрить другие экземпляры типа `ArticleService` в каждый из порожденных компонентов. Для этого достаточно перенести объявление свойства `providers` из определения модуля в определение класса `ArticleComponent`, как выделено ниже полужирным шрифтом. Оба полученных экземпляра можно проверить, наблюдая результаты, выводимые при вызове метода `console.log()` из конструктора класса `ArticleService`.

```
[app/article.module.ts]

import {NgModule} from '@angular/core';
import {ArticleComponent} from './article.component';

@NgModule({
  declarations: [
    ArticleComponent
  ],
  bootstrap: [
    ArticleComponent
  ],
  exports: [
    ArticleComponent
  ]
})
export class ArticleModule {}

[app/article.component.ts]

import {Component} from '@angular/core';
import {ArticleService} from './article.service';

@Component({
  selector: 'article',
  template: `
    <p>Article component!</p>
  `,
  providers: [
    ArticleService
  ]
})
export class ArticleComponent {
  constructor(private articleService_:ArticleService) {}
}
```

Получение экземпляра службы

Местоположение поставщиков означает также, что получение экземпляра службы привязано к времени жизни компонента. Для данного приложения это означает, что всякий раз, когда создается компонент, будет получен новый экземпляр службы, если она предоставляется.

Так, если попытаться создать порожденный компонент с предоставляемой в нем службой типа `ArticleService`, то всякий раз, когда произойдет создание компонента `ArticleComponent`, будет создана и новая служба типа `ArticleService`, как показано ниже. Можете сами убедиться, что новые экземпляры данной службы получаются всякий раз, когда вычисляется логическое значение `true` директивы `ngIf`, наблюдая результаты, выводимые при дополнительных вызовах метода `console.log()` из конструктора класса `ArticleService`.

```
[app/root.component.ts]

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

@Component({
  selector: 'root',
  template: `
    <h1>root component!</h1>
    <button (click)="toggle=!toggle">Toggle</button>
    <article></article>
    <article *ngIf="toggle"></article>
  `
})
export class RootComponent {}
```

См. также

- Рецепт "Внедрение простой службы в компонент" с пояснениями основ системы внедрения зависимостей в Angular 2.
- Рецепт "Назначение псевдонимов при внедрении служб с помощью свойств `useClass` и `useExisting`", в котором показано, как перехватывать запросы от поставщиков внедрения зависимостей.

Назначение псевдонимов при внедрении служб с помощью свойств `useClass` и `useExisting`

По мере усложнения разрабатываемого приложения может возникнуть ситуация, когда службами требуется воспользоваться в полиморфном стиле.

Так, в некоторых местах приложения может возникнуть потребность запросить службу А. Но конфигурация в какой-нибудь другой части приложения приведет к тому, что приложение фактически предоставит службу Б. В данном рецепте демонстрируется один полезный способ, но такое поведение позволяет расширить приложение несколькими способами.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/1109/**.

Подготовка

Допустим, что разработка приложения начинается с приведенного ниже каркаса.

Двойные службы

Начните с двух служб, представленных классами `ArticleService` и `EditorArticleService`, а также их интерфейса `ArticleSourceInterface`. В частности, класс `EditorArticleService` наследует от класса `ArticleService`, как показано ниже.

```
[app/article-source.interface.ts]

export interface ArticleSourceInterface {
  getArticle():Article
}

export interface Article {
  title:string,
  body:string,
  // Знак ? обозначает необязательное свойство
  notes?:string
}

[app/article.service.ts]

import {Injectable} from '@angular/core';
import {Article, ArticleSourceInterface}
        from './article-source.interface';

@Injectable()
export class ArticleService implements ArticleSourceInterface {
  private title_:string =
      "Researchers Determine Ham Sandwich Not Turing Complete";

  private body_:string =
          "Computer science community remains skeptical";

  getArticle():Article {
```

```
    return {
      title: this.title_,
      body: this.body_
    };
  }
}
```

[app/editor-article.service.ts]

```
import {Injectable} from '@angular/core';
import {ArticleService} from './article.service';
import {Article, ArticleSourceInterface}
        from './article-source.interface';

@Injectable()
export class EditorArticleService extends ArticleService
       implements ArticleSourceInterface {
  private notes_:string = "Swing and a miss!";

  constructor() {
    super();
  }

  getArticle():Article {
    // объединить объекты и возвратить соединенный объект
    return Object.assign(
      {},
      super.getArticle(),
      {
        notes: this.notes_
      });
  }
}
```

Единообразный компонент

Цель состоит в том, чтобы воспользоваться следующим компонентом и внедрить обе упомянутые выше службы в этом компоненте:

[app/article.component.ts]

```
import {Component} from '@angular/core';
import {ArticleService} from './article.service';
import {Article} from './article-source.interface';

@Component({
  selector: 'article',
  template: `
    <h2>{{article.title}}</h2>
    <p>{{article.body}}</p>
    <p *ngIf="article.notes">
```

```
      <i>Notes: {{article.notes}}</i>
    </p>
  `
})
export class ArticleComponent {
  article:Article;
  constructor(private articleService_:ArticleService) {
    this.article = articleService.getArticle();
  }
}
```

Реализация

При указании поставщиков в Angular 2 допускается объявлять ссылку с назначенным псевдонимом, обозначающую, какую именно службу следует предоставить, когда запрашивается один из определенных типов. А поскольку система внедрения зависимостей в Angular 2 последует вверх по иерархическому дереву компонентов, чтобы обнаружить поставщика, то объявить такой псевдоним можно, например, заключив компонент в родительский компонент, где этот псевдоним будет указан, как показано ниже.

```
[app/default-view.component.ts]

import {Component} from '@angular/core';
import {ArticleService} from './article.service';

@Component({
  selector: 'default-view',
  template: `
    <h3>Default view</h3>
    <ng-content></ng-content>
  `,
  providers: [ArticleService]
})
export class DefaultViewComponent {}

[app/editor-view.component.ts]

import {Component } from '@angular/core';
import {ArticleService} from './article.service';
import {EditorArticleService} from './editor-article.service';

@Component({
  selector: 'editor-view',
  template: `
    <h3>Editor view</h3>
    <ng-content></ng-content>
  `,
  providers: [
```

```
    {provide: ArticleService, useClass: EditorArticleService}
  ]
})
export class EditorViewComponent {}
```

Следует заметить, что оба приведенных выше класса действуют в качестве передаваемых компонентов. Помимо добавления заголовка, что делается лишь в целях пояснения данного рецепта, в этих классах указывается только поставщик безразлично к их содержимому.

Определив классы-оболочки, можно добавить их сначала в модуль приложения, а затем воспользоваться ими для создания двух экземпляров типа ArticleComponent, как выделено ниже полужирным шрифтом. Таким образом, заметки получаются в редакторской версии компонента ArticleComponent, тогда как в стандартной версии они отсутствуют.

```
[app/app.module.ts]

import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {RootComponent} from './root.component';
import {ArticleComponent} from './article.component';
import {DefaultViewComponent} from './default-view.component';
import {EditorViewComponent} from './editor-view.component';
import {ArticleComponent} from './article.component';
import {ArticleService} from './article.service';
import {EditorArticleService} from './editor-article.service';

@NgModule({
  imports: [
    BrowserModule
  ],
  declarations: [
    RootComponent,
    ArticleComponent,
    DefaultViewComponent,
    EditorViewComponent
  ],
  bootstrap: [
    RootComponent
  ]
})
export class AppModule {}

[app/root.component.ts]

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

@Component({
```

```
  selector: 'root',
  template: `
    <default-view>
      <article></article>
    </default-view>
    <hr />
    <editor-view>
      <article></article>
    </editor-view>
  `
})
export class RootComponent {}
```

Принцип действия

Вид службы, которую предполагалось внедрить, указывался в Angular 1 непосредственно в качестве параметра функции. Например, в результате вызова `function(Article)` внедрялась служба `Article`, а в результате вызова `function(User)` — служба `User`. Это приводило к таким побочным явлениям, как обеспечение минимальной защиты конструкторов, которым нужно было передавать массив символьных строк и соответствующих им функций вида `['Article', function(Article) {}]`.

Теперь все это уже не так. После регистрации поставщика в свойстве `useClass` применяется система сопоставления внедряемых зависимостей, состоящая из двух частей и принятая в Angular 2. К первой части относится поставщик маркеров, т.е. задаваемый в качестве параметра вид внедряемой службы. В данном случае в качестве параметра `private articleService_:ArticleService` задается маркер для запроса внедряемой службы типа `ArticleService`. Этот маркер сопоставляется с поставщиками из иерархии компонентов. Если маркер совпадает, вторая часть данной системы, т.е. сам поставщик, используется для внедрения экземпляра службы.

В действительности объявление `providers: [ArticleService]` является сокращением объявления `providers: [{provide:ArticleService, useClass: ArticleService}]`. Это удобное сокращение, поскольку оно почти всегда обозначает запрос класса сервера, который должен совпадать с внедряемым классом. Но в данном рецепте Angular 2 настраивается на распознавание маркера службы типа `ArticleService`, а следовательно, на применение поставщика службы типа `EditorArticleService`.

Дополнение

Внимательному читателю должно быть теперь ясно, что применение свойства `useClass` ограничено в том смысле, что оно не позволяет независимо контролировать, где именно предоставляется конкретная служба. Иными

словами, место, где перехватывается определение поставщика в свойстве useClass, служит также местом, где предоставляется заменяющий класс.

В данном примере свойство useClass оказывается пригодным потому, что оно дает идеальную возможность предоставить службу типа EditorArticleService в том же указанном месте, где она должна быть заменена службой типа ArticleService. Хотя нетрудно представить ситуацию, когда было бы желательно указать вид заменяющей службы, но внедрить ее выше по иерархии дерева компонентов. Ведь это позволило бы повторно пользоваться экземплярами службы вместо того, чтобы создавать новую службу для каждого объявления свойства useClass.

С этой целью можно воспользоваться свойством useExisting. Оно требует указать вид предоставляемой службы отдельно, но позволяет повторно пользоваться предоставленным ее экземпляром вместо того, чтобы создавать новый. Поэтому можете переконфигурировать только что созданное приложение, чтобы воспользоваться в нем свойством useExisting и предоставить оба рассматриваемых здесь вида служб на уровне компонента RootComponent.

Чтобы проверить правильность вашего представления о поведении служб, удвойте число компонентов Article и добавьте метод log() в конструктор класса ArticleService, чтобы убедиться в создании только одного экземпляра каждой службы, как показано ниже.

```
[app/root.component.ts]

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

@Component({
  selector: 'root',
  template: `
    <default-view>
      <article></article>
    </default-view>
    <editor-view>
      <article></article>
    </editor-view>
    <default-view>
      <article></article>
    </default-view>
    <editor-view>
      <article></article>
    </editor-view>
  `
})
export class RootComponent {}

[app/article.service.ts]
```

```
import {Injectable} from '@angular/core';
import {Article, ArticleSourceInterface}
        from './article-source.interface';

@Injectable()
export class ArticleService implements ArticleSourceInterface {
  private title_:string =
     "Researchers Determine Ham Sandwich Not Turing Complete";
  private body_:string =
     "Computer science community remains skeptical";
  constructor() {
    console.log('Instantiated ArticleService!');
  }
  getArticle():Article {
    return {
      title: this.title_,
      body: this.body_
    };
  }
}
```

[app/editor-view.component.ts]

```
import {Component} from '@angular/core';
import {ArticleService} from './article.service';
import {EditorArticleService} from './editor-article.service';

@Component({
  selector: 'editor-view',
  template: `
    <h3>Editor view</h3>
    <ng-content></ng-content>
  `,
  providers: [
    {provide: ArticleService, useExisting: EditorArticleService}
  ]
})
export class EditorViewComponent {}
```

[app/default-view.component.ts]

```
import {Component} from '@angular/core';
import {ArticleService} from './article.service';

@Component({
  selector: 'default-view',
  template: `
    <h3>Default view</h3>
    <ng-content></ng-content>
  `
  // удаленные поставщики
```

```
})
export class DefaultViewComponent {}
```

Если в данной конфигурации применяется свойство `useClass`, то в конечном счете создается один экземпляр службы типа `ArticleService` и два экземпляра службы типа `EditorArticleService`. А после замены на свойство `useExisting` создается лишь один экземпляр каждой из этих служб.

Таким образом, в переконфигурированной версии данного рецепта рассматриваемое здесь приложение выполняет следующие действия.

- На уровне компонента `RootComponent` предоставляется служба типа `EditorArticleService`.
- На уровне компонента `EditorViewComponent` маркеры внедрения службы типа `ArticleService` переадресовываются службе типа `EditorArticleService`.
- На уровне компонента `ArticleComponent` служба типа `ArticleService` внедряется по своему маркеру.

Реорганизация кода с помощью директивных поставщиков

Если рассматриваемая здесь реализация покажется вам неуклюжей и многословной, значит, вы верно понимаете суть дела. Промежуточные компоненты вполне справляются со своими обязанностями, но эти обязанности не выходят за рамки вставки объявления промежуточного поставщика в виде прокладки. Вместо того чтобы заключать объявление поставщика в оболочку компонента, соответствующий оператор можно перенести в директиву и тем самым избавиться от обоих компонентов представлений, как выделено ниже полужирным шрифтом. В итоге приложение будет работать точно так же!

```
[app/root.component.ts]

import {Component} from '@angular/core';
import {ArticleComponent} from './ article.component';
import {ArticleService} from './article.service';
import {EditorArticleService} from './editor-article.service';

@Component({
  selector: 'root',
  template: `
    <article></article>
    <article editor-view></article>
    <article></article>
    <article editor-view></article>
  `
})
export class RootComponent {}
```

```
[app/editor-view.directive.ts]

import {Directive } from '@angular/core';
import {ArticleService} from './article.service';
import {EditorArticleService} from './editor-article.service';

@Directive({
  selector: '[editor-view]',
  providers: [
    {provide: ArticleService, useExisting: EditorArticleService}
  ]
})
export class EditorViewDirective {}

[app/app.module.ts]

import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {RootComponent} from './root.component';
import {ArticleComponent} from './article.component';
import {DefaultViewComponent} from './default-view.component';
import {EditorViewDirective} from './editor-view.directive';
import {ArticleComponent} from './article.component';
import {ArticleService} from './article.service';
import {EditorArticleService} from './editor-article.service';

@NgModule({
  imports: [
  BrowserModule
  ],
  declarations: [
    RootComponent,
    ArticleComponent,
    DefaultViewComponent,
    EditorViewDirective
  ],
  providers: [
    ArticleService,
    EditorArticleService
  ],
  bootstrap: [
    RootComponent
  ]
})
export class AppModule {}
```

См. также

- Рецепт "Внедрение простой службы в компонент" с пояснениями основ системы внедрения зависимостей в Angular 2.

- Рецепт "Контроль над созданием экземпляра службы и внедрением средствами класса NgModule", в котором дается обширный обзор, каким образом в Angular 2 строятся иерархии поставщиков с помощью модулей.
- Рецепт "Внедрение значения в виде службы с помощью свойства useValue и класса OpaqueToken", в котором показано, как пользоваться маркерами внедряемых зависимостей для внедрения обобщенных объектов.
- Рецепт "Создание настраиваемой поставщиком службы с помощью свойства useFactory" с подробным описанием процесса установки фабрики служб для формирования определений настраиваемых служб.

Внедрение значения в виде службы с помощью свойства useValue и класса OpaqueToken

В версии Angular 1 предоставлялся на выбор обширный ряд типов служб, которыми можно пользоваться в приложении. К их числу относятся типы служб, позволяющие внедрять статическое значение вместо экземпляра службы. Эта удобная возможность сохранилась и в версии Angular 2.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/3032/**.

Подготовка

Допустим, имеется следующая заготовка приложения:

```
[app/app.module.ts]

import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {RootComponent} from './root.component';
import {ArticleComponent} from './article.component';

@NgModule({
  imports: [
    BrowserModule
  ],
  declarations: [
    RootComponent,
    ArticleComponent
  ],
  bootstrap: [
```

```
    RootComponent
  ]
})
export class AppModule {}

[app/root.component.ts]

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

@Component({
  selector: 'root',
  template: `
    <article></article>
  `
})
export class RootComponent {}

[app/article.component.ts]

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

@Component({
  selector: 'article',
  template: `
    <img src="{{logoUrl}}">
    <h2>Fool and His Money Reunited at Last</h2>
    <p>Author: Jake Hsu</p>
  `
})
export class ArticleComponent {}
```

Реализация

Несмотря на то что для внедрения отдельного значения формальное объявление класса службы и обозначение декоратора `@Injectable` больше не требуется, сопоставление маркера с поставщиком по-прежнему необходимо. А поскольку класс, предназначенный для приведения типа внедряемого значения больше недоступен, то вместо него должно действовать нечто иное.

Это затруднение разрешается в Angular 2 с помощью класса `OpaqueToken`. Модуль, который представляет этот класс, позволяет создавать бесклассовый маркер, которым можно воспользоваться для соединения внедряемого значения в паре с аргументом конструктора. Наряду с этим можно воспользоваться свойством `useValue`, непосредственно предоставляющим любое свое содержимое в качестве внедряемого значения.

Определите маркер, указав однозначную символьную строку в его конструкторе следующим образом:

[app/logo-url.token.ts]

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

export const LOGO_URL = new OpaqueToken('logo.url');
```

Внедрите этот маркер в определение модуля приложения обычным образом. Но при этом необходимо обозначить, на что именно он будет указывать после сопоставления. В данном случае он должен указывать на символьную строку изображения в URL:

[app/app.module.ts]

```
import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {RootComponent} from './root.component';
import {ArticleComponent} from './article.component';
import {LOGO_URL} from './logo-url.token';

@NgModule({
  imports: [
    BrowserModule
  ],
  declarations: [
    RootComponent,
    ArticleComponent
  ],
  providers: [
    {provide: LOGO_URL, useValue:
              'https://angular.io/resources/images/
              ↳logos/standard/logo-nav.png'}
  ],
  bootstrap: [
    RootComponent
  ]
})
export class AppModule {}
```

И, наконец, данный маркер можно внедрить в компонент. Но поскольку внедряется нечто, еще не определенное с помощью декорации `@Injectable()`, то в конструкторе следует использовать декорацию `@Inject()`, чтобы сообщить Angular, что это должно быть сделано путем внедрения зависимостей. Кроме того, внедрение не присоединяется к ссылке `this` компонента автоматически, а следовательно, это придется также сделать вручную, как показано ниже. В конечном итоге изображение должно быть благополучно отображено в окне браузера!

```
[app/article.component.ts]

import {Component, Inject} from '@angular/core';
import {LOGO_URL} from './logo-url.token';

@Component({
  selector: 'article',
  template: `
    <img src="{{logoUrl}}">
    <h2>Fool and His Money Reunited at Last</h2>
    <p>Author: Jake Hsu</p>
  `
})
export class ArticleComponent {
  logoUrl:string;
  constructor(@Inject(LOGO_URL) private logoUrl_) {
    this.logoUrl = logoUrl_;
  }
}
```

Принцип действия

Класс `OpaqueToken` позволяет пользоваться неклассовыми типами в схеме поставщиков, ориентированной на классы и внедренной в Angular 2. Именно этот класс будет использоваться в каркасе внедрения зависимостей при попытке сопоставить маркеры внедрения зависимостей с объявлениями поставщиков. Это дает возможность более широко употреблять внедрение зависимостей повсюду в приложении, поскольку значение любого типа можно теперь предоставить везде, где бы ни внедрялась служба определенного типа.

Дополнение

Другой удобный способ внедрения значений дает возможность имитации службы. Допустим, требуется определить стандартную службу-заглушку, которую следует затем переопределить с помощью явного поставщика, чтобы активизировать полезное поведение. В подобных случаях нетрудно представить себе стандартный элемент статьи, который можно было бы по-другому сконфигурировать через директиву, повторно используя тот же самый компонент. Таким образом, в определенном ниже компоненте типа `ArticleComponent` будет использоваться имитирующая служба, когда директива не присоединяется, а когда она присоединяется, то конкретная служба.

```
[app/root.component.ts]

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

```
@Component({
  selector: 'root',
  template:
    <article></article>
    <article editor-view></article>
  `
})
export class RootComponent {}

[app/editor-article.service.ts]

import {Injectable} from '@angular/core';

export const MockEditorArticleService = {
  getArticle: () => ({
    title: "Mock title",
    body: "Mock body"
  })
};

@Injectable()
export class EditorArticleService {
  private title_:string =
      "Prominent Vegan Embroiled in Scrambled Eggs Scandal";
  private body_:string =
      "Tofu Farming Alliance retracted their endorsement.";

  getArticle() {
    return {
      title: this.title_,
      body: this.body_
    };
  }
}

[app/editor-view.directive.ts]

import {Directive} from '@angular/core';
import {EditorArticleService} from './editor-article.service';

@Directive({
  selector: '[editor-view]',
   providers: [EditorArticleService]
})
export class EditorViewDirective {}

[app/article.component.ts]

import {Component, Inject} from '@angular/core';
import {EditorArticleService} from './editor-article.service';
```

```
@Component({
  selector: 'article',
  template: `
    <h2>{{title}}</h2>
    <p>{{body}}</p>
  `
})
export class ArticleComponent {
  title:string;
  body:string;
  constructor(
        private editorArticleService_:EditorArticleService) {
    let article = editorArticleService_.getArticle();
    this.title = article.title;
    this.body = article.body;
  }
}
```

См. также

- Рецепт "Контроль над созданием экземпляра службы и внедрением средствами класса NgModule" с обширным обзором, каким образом в Angular 2 строятся иерархии поставщиков с помощью модулей.
- Рецепт "Назначение псевдонимов при внедрении служб с помощью свойств useClass и useExisting", в котором демонстрируется, как перехватывать запросы от поставщиков внедрения зависимостей.
- Рецепт "Создание настраиваемой поставщиком службы с помощью свойства useFactory" с подробным описанием процесса установки фабрики служб для формирования определений настраиваемых служб.

Создание настраиваемой поставщиком службы с помощью свойства useFactory

Еще одним расширением внедрения зависимостей в Angular 2 является возможность пользоваться фабриками при определении иерархии поставщиков. Фабрика поставщиков позволяет принимать входные данные, выполнять произвольные операции, чтобы сконфигурировать поставщика и возвратить экземпляр этого поставщика для внедрения.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/0049/**.

Подготовка

Начните снова с установки компонента, состоящего из двойной службы и статьи и представленного ранее в рецепте "Назначение псевдонимов при внедрении служб с помощью свойств `useClass` и `useExisting`".

Реализация

Фабрики поставщиков в Angular 2, как подразумевает их название, являются функциями, возвращающими поставщика. Фабрика может быть указана в отдельном файле и доступна по ссылке в свойстве `useFactory`.

Начните с объединения двух служб в единую службу, для конфигурирования которой должен быть вызвав соответствующий метод, как показано ниже.

```
[app/article.service.ts]

import {Injectable} from '@angular/core';

@Injectable()
export class ArticleService {
  private title_:string =
          "Flying Spaghetti Monster Sighted";
  private body_:string =
          "Adherents insist we are missing the point";
  private notes_:string = "Spot on!";
  private editorEnabled_:boolean = false;

  getArticle():Object {
    var article = {
      title: this.title_,
      body: this.body_
    };
    if (this.editorEnabled_) {
      Object.assign(article, article, {
        notes: this.notes_
      });
    }
    return article;
  }

  enableEditor():void {
    this.editorEnabled_ = true;
  }
}
```

Определение фабрики

Цель состоит в том, чтобы сконфигурировать данную службу таким образом, чтобы вызывать метод `enableEditor()`, исходя из состояния логического признака. Это можно сделать с помощью фабрик поставщиков. Поэтому определите фабрику в отдельном файле следующим образом:

```
[app/article.factory.ts]

import {ArticleService} from './article.service';

export function articleFactory(
       enableEditor?:boolean):ArticleService {
  return (articleService:ArticleService) => {
    if (enableEditor) {
      articleService.enableEditor();
    }
    return articleService;
  }
}
```

Внедрение маркера типа `OpaqueToken`

Превосходно! А далее необходимо переконфигурировать компонент `ArticleComponent`, чтобы внедрить маркер вместо требующейся службы:

```
[app/article.token.ts]

import {OpaqueToken} from '@angular/core';

export const ArticleToken = new OpaqueToken('app.article');

[app/article.component.ts]

import {Component, Inject} from '@angular/core';
import {ArticleToken} from './article.token';

@Component({
  selector: 'article',
  template: `
    <h2>{{article.title}}</h2>
    <p>{{article.body}}</p>
    <p *ngIf="article.notes">
      <i>Notes: {{article.notes}}</i>
    </p>
  `
})
export class ArticleComponent {
  article:Object;
  constructor(@Inject(ArticleToken) private articleService_) {
```

```
    this.article = articleService_.getArticle();
  }
}
```

Создание директив поставщиков с помощью свойства useFactory

И, наконец, необходимо определить директивы, чтобы указать порядок применения данной фабрики и внедрения этих директив в приложение, как показано ниже. Таким образом, должны быть обнаружены обе версии компонента ArticleComponent.

[app/default-view.directive.ts]

```
import {Directive} from '@angular/core';
import {ArticleService} from './article.service';
import {articleFactory} from './article.factory';
import {ArticleToken} from './article.token';

@Directive({
  selector: '[default-view]',
  providers: [
    {provide: ArticleToken,
      useFactory: articleFactory(),
      deps: [ArticleService]
    }
  ]
})
export class DefaultViewDirective {}
```

[app/editor-view.directive.ts]

```
import {Directive} from '@angular/core';
import {ArticleService} from './article.service';
import {articleFactory} from './article.factory';
import {ArticleToken} from './article.token';

@Directive({
  selector: '[editor-view]',
  providers: [
    {
      provide: ArticleToken,
      useFactory: articleFactory(true),
      deps: [ArticleService]
    }
  ]
})
export class EditorViewDirective {}
```

```
[app/root.component.ts]

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

@Component({
  selector: 'root',
  template: `
    <article default-view></article>
    <article editor-view></article>
  `
})
export class RootComponent {}
```

Принцип действия

Компонент статьи переопределяется таким образом, чтобы использовать маркер вместо внедрения службы. При наличии маркера Angular может обойти иерархическое дерево компонентов, чтобы обнаружить место, где предоставляется этот маркер. В директивах объявляется, что маркер сопоставляется с фабрикой поставщиков, т.е. с методом, вызываемым для возврата конкретного поставщика. Именно свойство `useFactory` и сопоставляет маркер с фабричным методом, а свойство `deps` — с зависимостями служб, которыми обладает фабрика.

Дополнение

Важное отличие, которое следует признать в данный момент, заключается в том, что все эти конфигурации фабрик происходят прежде получения экземпляров любых компонентов. После установки в декорации класса, где определяются поставщики, будет вызван фабричный метод.

См. также

- Рецепт "Контроль над созданием экземпляра службы и внедрением средствами класса `NgModule`", в котором дается обширный обзор, каким образом в Angular 2 строятся иерархии поставщиков с помощью модулей.
- Рецепт "Назначение псевдонимов при внедрении служб с помощью свойств `useClass` и `useExisting`", в котором показано, как перехватывать запросы от поставщиков внедрения зависимостей.
- Рецепт "Внедрение значения в виде службы с помощью свойства `useValue` и класса `OpaqueToken`", где показывается, как пользоваться маркерами внедряемых зависимостей для внедрения обобщенных объектов.

Глава 8

Организация приложений и управление ими

В этой главе даются следующие рецепты.

- Содержимое файла `package.json` для минимально жизнеспособного приложения Angular 2
- Конфигурирование TypeScript для минимально жизнеспособного приложения Angular 2
- Транспиляция в браузере средствами библиотеки SystemJS
- Составление файлов приложений для минимально жизнеспособного приложения Angular 2
- Перенос минимально жизнеспособного приложения Angular 2 в сборку Webpack
- Внедрение прокладок и полифилов в Webpack
- Формирование HTML-разметки с помощью подключаемого модуля `html-webpack-plugin`
- Установка приложения Angular с помощью интерфейса командной строки

Введение

К честолюбивым целям проекта Angular 2 относится использование другого языка с иным синтаксисом и конструкциями, а также обеспечение высокой эффективности и модульности. Для разработчиков веб-приложений это означает, что процесс сопровождения приложений Angular 2 может усложниться.

Конечная цель состоит в том, чтобы эффективно предоставлять HTML-документы, таблицы CSS и сценарии JavaScript для обработки в веб-браузере,

а также упростить разработку исходных компонентов для этих статических файлов. Достичь этой конечной цели можно разными путями, и было бы пустой тратой времени посвятить данному предмету отдельную главу.

Вместо этого здесь представлен ряд проверенных способов организации приложений Angular 2, отражающих самые распространенные и эффективные стратегии. В этой главе также демонстрируется порядок написания и расширения минимально жизнеспособного приложения Angular 2. Некоторым читателям он может показаться слишком простым и элементарным. Но в большинстве проектов Quickstart или каркасов генерирования исходного кода предоставляется лишь информационное хранилище и несколько команд для практического применения, причем эти команды выполняются без пояснения, что и как они делают! Из этой главы вы узнаете, как написать приложение на Angular 2 с самого начала, применяя вспомогательные пакеты и инструментальные средства, а также ознакомитесь с причинами, по которым выбраны все эти способы и средства.

Содержимое файла `package.json` для минимально жизнеспособного приложения Angular 2

Когда речь заходит о минимально жизнеспособном приложении Angular 2, то ничто не оказывается так близко к уровню среды выполнения, как файлы конфигурации. В данном случае речь идет о двух файлах конфигурации (`package.json` и `tsconfig.json`), где определяется порядок управления файлами и установки пакетов с помощью диспетчера пакетов `npm` и процессы их запуска.

Отчасти данный рецепт может послужить кратким напоминанием для тех разработчиков, у которых имеется опыт работы с диспетчером `npm` и его средствами. Однако даже самую простую конфигурацию проекта в Angular 2 можно структурировать таким образом, чтобы стали совершенно понятными более сложные конфигурации, созданные на этих основах.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **`http://ngcookbook.herokuapp.com/1332/`**.

Подготовка

Для проработки данного рецепта придется установить платформу Node.js, а также создать пустой каталог для текущего проекта. Создайте две следующие заготовки файлов конфигурации в корневом каталоге данного проекта:

```
[package.json]
{
  "name": "angular2-minimum-viable-application"
}

[tsconfig.json]
{
  "compilerOptions": {
}
}
```

Чтобы просто и быстро убедиться, что диспетчер npm установлен правильно и готов к применению, запустите следующую команду:

```
npm --version
```

Эта команда должна выдать номер версии, если все установлено правильно.

Реализация

Начните с файла `package.json`. Он служит для конфигурации минимально жизнеспособного приложения и состоит из следующих разделов.

- **`dependencies`**. Здесь перечислены целевые ресурсы пакета, от которых непосредственно зависит работа приложения.
- **`devDependencies`**. В этом разделе перечислены целевые ресурсы пакета, которые требуются локальной среде по самым разным причинам, включая компиляцию, выполнение тестов или статический анализ.
- **`scripts`**. Здесь указаны специально определяемые утилиты командной строки, выполняемые средствами диспетчера npm.

Раздел `dependencies` из файла `package.json`

Прежде всего необходимо определить все зависимости, которые потребуются в приложении. К их числу относятся зависимости от базовых модулей Angular 2, находящихся в каталоге `node_modules/@angular`, а также перечисленные ниже библиотечные зависимости.

- **`core-js`**. Этот полифил требуется для поддержки синтаксиса стандарта ES6 языка JavaScript, который используется в компиляторе языка TypeScript, включая классы `Set`, `Promise` и `Map`.
- **`reflect-metadata`**. Этот полифил требуется для реализации интерфейса Reflect Metadata API. Он позволяет применять декораторы, не входящие в стандартную спецификацию языка TypeScript, например, декоратор `@Component`.

- **rxjs**. Предназначается для библиотеки наблюдаемых объектов ReactiveX на языке JavaScript. В каркасе Angular 2 применяются собственные наблюдаемые объекты, напрямую зависящие от данной библиотеки.
- **SystemJS**. Это динамический загрузчик модулей, требующийся в данном проекте для достижения следующих целей: импортировать и сопоставить все исходные файлы, а также разрешить объявления `import/export` стандарта ES6 языка JavaScript.
- **zonejs**. Это библиотека ZoneJS, предоставляющая возможность пользоваться в Angular 2 контекстами асинхронного выполнения. Каркас Angular 2 напрямую зависит от данной библиотеки.

В итоге файл `package.json` должен быть дополнен следующим содержимым:

```
[package.json]

{
  "name": "angular2-minimum-viable-application",
  "dependencies": {
    "@angular/common": "2.0.0",
    "@angular/compiler": "2.0.0",
    "@angular/core": "2.0.0",
    "@angular/platform-browser": "2.0.0",
    "@angular/platform-browser-dynamic": "2.0.0",
    "core-js": "^2.4.1",
    "reflect-metadata": "^0.1.3",
    "rxjs": "5.0.0-beta.12",
    "systemjs": "0.19.27",
    "zone.js": "^0.6.23"
  }
}
```

Раздел devDependencies из файла package.json

Далее требуется указать следующие зависимости в разделе `devDependencies`:

Напомним, что в диспетчере `npm` к типу **devDependencies** относятся зависимости, характерные для среды разработки. Ими можно пользоваться при написании сценариев, чтобы отличать пакеты, включаемые в рабочий комплект, от пакетов, не включаемых в него.

- **lite-server**. Это простой файловый сервер, предназначенный для локального тестирования данного приложения. Его можно заменить любыми другими простыми файловыми серверами.

- **typescript**. Это компилятор языка TypeScript.
- **concurrently**. Это простая утилита командной строки, предназначенная для одновременного выполнения команд из сценария в npm.

В итоге файл package.json должен быть дополнен следующим содержимым:

[package.json]

```
{
  "name": "angular2-minimum-viable-application",
  "dependencies": {
    "@angular/common": "2.0.0",
    "@angular/compiler": "2.0.0",
    "@angular/core": "2.0.0",
    "@angular/platform-browser": "2.0.0",
    "@angular/platform-browser-dynamic": "2.0.0",
    "core-js": "^2.4.1",
    "reflect-metadata": "^0.1.3",
    "rxjs": "5.0.0-beta.12",
    "systemjs": "0.19.27",
    "zone.js": "^0.6.23"
  },
  "devDependencies": {
    "concurrently": "^2.2.0",
    "lite-server": "^2.2.2",
    "typescript": "^2.0.2"
  }
}
```

Раздел scripts из файла package.json

Наконец, необходимо создать сценарии, предназначенные для формирования скомпилированных файлов и запуска сервера разработки. Эти сценарии выделены ниже полужирным шрифтом в разделе scripts из файла package.json.

[package.json]

```
{
  "name": "angular2-minimum-viable-application",
  "scripts": {
    "lite": "lite-server",
    "postinstall": "npm install -S @types/node @types/core-js",
    "start": "tsc && concurrently 'npm run tsc:w' 'npm run lite'",
    "tsc": "tsc",
    "tsc:w": "tsc -w"
  },
```

```
  "dependencies": {
    "@angular/common": "2.0.0",
    "@angular/compiler": "2.0.0",
    "@angular/core": "2.0.0",
    "@angular/platform-browser": "2.0.0",
    "@angular/platform-browser-dynamic": "2.0.0",
    "core-js": "^2.4.1",
    "reflect-metadata": "^0.1.3",
    "rxjs": "5.0.0-beta.12",
    "systemjs": "0.19.27",
    "zone.js": "^0.6.23"
  },
  "devDependencies": {
    "concurrently": "^2.2.0",
    "lite-server": "^2.2.2",
    "typescript": "^2.0.2"
  }
}
```

Каждый из этих сценариев служит определенной цели, но большинство из них не требуется вызывать вручную. Ниже приведено краткое описание каждого из них.

- **lite**. Запускает экземпляр файлового сервера `lite-server`.
- **postinstall**. Определяет перехватчик, выполняемый после команды **npm install**. В данном случае после установки в `npm` всех проектных зависимостей требуется установить файлы объявлений для тех модулей, где они отсутствуют. Всякий раз, когда выполняется сценарий, в `npm` сначала проверяются сценарии с префиксами **pre** и **post** в их именах, а затем они выполняются до или после этого сценария соответственно. В данном рецепте сценарий `prelite` будет выполнен прежде сценария `lite`, а сценарий `postlite` — после него.
- **start**. Определяет значение, задаваемое по умолчанию в команде **npm start**. Этот сценарий запускает компилятор языка TypeScript один раз до полного завершения, а затем вызывает наблюдатель компилятора TypeScript и одновременно запускает сервер разработки. Слово **start** зарезервировано в `npm` для данного сценария, поэтому выполнять команду **npm run start** не нужно, хотя она вполне работоспособна.
- **tsc**. Запускает компилятор языка TypeScript. Этот компилятор читает свои параметры настройки из файла конфигурации `tsconfig.json`, находящегося в том же самом каталоге.
- **tsc:w**. Устанавливает наблюдатель файлов на перекомпиляцию после внесения изменений в файл.

См. также

- Рецепт "Конфигурирование TypeScript для минимально жизнеспособного приложения Angular 2", в котором поясняется, как настроить компиляцию на поддержку проекта Angular 2.
- Рецепт "Транспиляция в браузере средствами библиотеки SystemJS", в котором демонстрируется, как воспользоваться библиотекой SystemJS, чтобы связать вместе нескомпилированные статические файлы.
- Рецепт "Составление файлов приложений для минимально жизнеспособного приложения Angular 2", где постепенно разъясняется, как написать очень простое приложение Angular 2 с самого начала.
- Рецепт "Перенос минимально жизнеспособного приложения Angular 2 в сборку Webpack", где описывается порядок внедрения Webpack в процесс написания приложения в Angular.
- Рецепт "Внедрение прокладок и полифилов в Webpack" с описанием удобного способа управления зависимостями полифилов в Angular 2.
- Рецепт "Формирование HTML-разметки с помощью подключаемого модуля `html-webpack-plugin`", в котором показано, как настроить пакет `npm` на автоматическое добавление скомпилированных файлов в разметку HTML-документа.
- Рецепт "Установка приложения Angular с помощью интерфейса командной строки" с описанием особенностей применения интерфейса командной строки, что он дает разработчикам и каково назначение его отдельных частей.

Конфигурирование TypeScript для минимально жизнеспособного приложения Angular 2

Чтобы воспользоваться TypeScript вместе с Angular 2, необходимо принять во внимание два следующих главных соображения: взаимная совместимость модулей и компиляция. Оба эти соображения необходимо учитывать для того, чтобы выбирать файлы приложения с расширением **`.ts`**, сочетать их с внешними библиотечными файлами и выводить файлы, совместимые с целевым устройством.

Компилятор TypeScript существует в виде готового к применению пакета `npm`. Но необходимо каким-то образом указать ему, как взаимодействовать с исходными файлами и модулями из других пакетов, которые требуется использовать в этих модулях.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/1053/**.

Подготовка

Сначала необходимо выполнить инструкции, упоминавшиеся в предыдущем рецепте. В результате будет создан каркас, необходимый для определения конфигурации TypeScript.

Реализация

Чтобы сконфигурировать TypeScript, необходимо добавить файлы объявлений в несовместимые модули и сформировать файл конфигурации, где указывается порядок работы компилятора.

Файлы объявлений

Файлы объявлений TypeScript существуют для того, чтобы указать вид библиотеки. Такие файлы можно выявить по суффиксу **`.d.ts`**. Эти файлы уже находятся в стандартном месте расположения в большинстве пакетов `npm` и других библиотек JavaScript. Это позволяет компилятору TypeScript их обнаружить и выяснить, каким образом следует интерпретировать эти библиотеки. А те библиотеки, в которые эти файлы не входят, должны быть им предоставлены. И, к счастью, сообщество разработчиков открытого исходного кода уже предоставляет немало подобных файлов.

Файлы объявлений, применяемые в данном проекте, отсутствуют в библиотеках `node` и `core-js`. Начиная с версии TypeScript 2.0, можно самостоятельно установить файлы объявлений для этих библиотек непосредственно через `npm`, указав параметр **`-S`** в приведенной ниже команде, где он означает их сохранение в файле конфигурации `package.json`. Удобным местом для помещения этой команды служит сценарий `postinstall`.

```
npm install -S @types/node @types/core-js
```

Файл конфигурации `tsconfig.json`

Компилятор TypeScript будет искать файл конфигурации `tsconfig.json`, чтобы определить порядок компиляции исходных файлов TypeScript в данном каталоге. Этот файл конфигурации обычно не требуется, поскольку компилятор TypeScript всегда может использовать стандартные настройки. Тем не менее он потребуется для того, чтобы точно указать порядок формирования файлов с расширением **`*.js`** и **`*.map.js`**. Поэтому внесите в файл конфигурации `tsconfig.json` изменения, выделенные ниже полужирным шрифтом.

[tsconfig.json]

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

Как и следовало ожидать, в свойстве `compilerOptions` указываются настройки, которые должны использоваться в компиляторе при обработке исходных файлов TypeScript в процессе компиляции. При отсутствии свойства `files` компилятор TypeScript обойдет всю структуру каталога проекта в поисках файлов с расширением **`*.ts`** и **`*.map.tsx`**.

Все свойства `compilerOptions` могут быть в равной степени указаны в качестве параметров командной строки. Но это можно сделать более организованным способом в файле конфигурации `tsconfig.json`. Ниже дается краткое описание отдельных свойств, употребляемых в данном проекте.

- **`target`**. Обозначает стандарт ECMAScript для вывода результатов компиляции. Ради обширной совместимости с браузерами здесь благоразумно оставить устанавливаемый по умолчанию стандарт ES5. Напомним, что ECMAScript — это спецификация, на основании которой построен язык JavaScript. Самым последним и законченным является стандарт ES6, называемый также спецификацией ES2015, но он поддерживается полностью пока еще не во всех браузерах. Поэтому компилятор TypeScript скомпилирует языковые конструкции стандарта ES6 (например, `class` и `Promise`) в их эквивалентную реализацию в стандарте ES5.
- **`module`**. Обозначает порядок работы с модулями в выходных файлах. В связи с тем что модули стандарта ES6 могут быть совместимы не со всеми браузерами, компилятору TypeScript придется преобразовать их модульную систему, пригодную для работы в текущем браузере. Поэтому здесь стоит указать библиотеку CommonJS. Модульный стиль библиотеки CommonJS подразумевает определение всех операций экспорта в едином модуле как свойств одного "экспортного" объекта. В компиляторе TypeScript поддерживаются также модули AMD (в стиле библиотеки `require.js`), UMD и, конечно, SystemJS, и при этом сохраняется присущий им модульный стиль стандарта ES6. Впрочем, более подробное рассмотрение модулей выходит за рамки данного рецепта.

- **moduleResolution**. Определяет порядок разрешения путей к модулям. Это свойство не настолько важно, чтобы понимать особенности стратегии, применяемой для разрешения путей к модулям, но устанавливаемое в нем значение `node` задает надлежащий формат вывода результатов компиляции.
- **emitDecoratorMetadata и experimentalDecorators**. Дают компилятору TypeScript возможность обрабатывать применяемые в Angular 2 декораторы. Напомним, что для поддержки экспериментальных декораторов следует добавить библиотеку `reflectmetadata`. Эти свойства служат точками привязки для компилятора TypeScript.
- **noImplicitAny**. Определяет, подлежат ли исходные файлы TypeScript типизации. Если здесь установлено логическое значение `true`, то в отсутствие любой типизации в данном проекте возникнет соответствующая ошибка. Не прекращается горячая полемика, стоит ли вообще устанавливать данное свойство, хотя принудительная типизация объектов, очевидно, полезна для предотвращения ошибок, которые могут возникнуть из-за неоднозначности в кодовых базах. Если требуется рассмотреть конкретный пример, когда компилятор выдаст ошибку несоблюдения типов, установите логическое значение `true` в данном свойстве и добавьте оператор `constructor (foo) {}` в класс `AppComponent`. В итоге должна появиться ошибка компиляции, извещающая, что параметр `foo` данного конструктора нетипизирован.

Принцип действия

Выполнение следующей команды приведет к запуску компилятора TypeScript из командной строки на корневом уровне каталога данного проекта:

```
npm run tsc
```

Компилятор обнаружит файл конфигурации `tsconfig.json`, если он существует в указанном месте, а иначе он обратится к стандартным настройкам. Настройки в этом файле дают компилятору указания, как обрабатывать файлы и проверять их на достоверность. Именно здесь и вступают в действие все сделанные ранее настройки.

Компилятор TypeScript не выполняет исходный код и не анализирует, что именно этот код фактически делает, но в то же время он может выявить момент, когда разные части приложения пытаются взаимодействовать ненадлежащим образом. В файле с расширением **`.d.ts`** и объявлениями для модуля задается порядок проверки в TypeScript интерфейса, где этот модуль станет доступным для использования после импорта.

Допустим, что во внешнем модуле `auth` содержится класс `User`. В таком случае этот класс можно импортировать следующим образом:

```
import {User} from './auth';
```

Благодаря добавлению файла объявлений в импортируемый модуль компилятор TypeScript может проверить, существует ли класс `User`. И он поведет себя так, как будто предпринимается попытка обратиться к данному классу в локальном модуле. Если обнаружится несоответствие, то в процессе компиляции будет выдана ошибка.

Компиляция

Вам, вероятно, не раз приходилось делать нечто подобное, хотя это зависит от вашего предыдущего опыта. Как и многие другие каркасы, Angular 2 действует, исходя из того, что языковых средств JavaScript в том виде, в каком они существуют, явно недостаточно для написания качественного кода. Определение "качественный" носит здесь субъективный характер, но все каркасы, требующие компиляции, призваны расширить или модифицировать возможности JavaScript в той или иной форме.

Тем не менее на всех платформах, где должны выполняться приложения (в данном случае — в веб-браузерах), имеется лишь среда выполнения JavaScript-кода из нескомпилированных исходных файлов. Вам не удастся расширить возможности браузера, чтобы тот мог обрабатывать полезную информацию или как то повлиять на доставляемый скомпилированный двоичный код, и поэтому файлы, отправляемые вами клиенту, должны соблюдать правила, принятые в браузере.

По определению и замыслу язык TypeScript является надмножеством, расширяющим стандарт ES6, но подобные расширения нельзя применять в самом браузере. И даже в настоящее время стандарт ES6 языка JavaScript поддерживается полностью не во всех браузерах. Поэтому вполне благоразумно привести исходный код TypeScript к спецификации ES5.1, которая стала стандартом ECMAScript и поддерживается во всех современных браузерах. Добиться подобного результата можно двумя перечисленными ниже способами.

- Отправить код TypeScript клиенту в исходном виде. Имеются встраиваемые в браузеры библиотеки, способные компилировать и выполнять на стороне клиента получающийся в итоге и совместимый со спецификацией ES5.1 код как обычный код JavaScript. Такой способ упрощает разработку веб-приложений, поскольку на стороне сервера требуется лишь обслуживание файлов. Но при этом вычисление переносится на сторону клиента, что приводит к снижению производительности, а следовательно, считается порочной нормой практики для разработки рабочих приложений.

- Скомпилировать исходный код TypeScript в код JavaScript, прежде чем отправить его клиенту. В подавляющем большинстве рабочих приложений применяется именно такой способ. Это объясняется, в частности, тем, что статические файлы нередко получаются из сети доставки содержимого (CDN) или статического каталога, и поэтому описательную кодовую базу TypeScript целесообразно скомпилировать в исходные файлы JavaScript как часть выпуска, а затем предоставить их клиенту.

Если проанализировать код JavaScript, получаемый в результате компиляции исходного кода TypeScript, то он окажется весьма грубым и неудобочитаемым. Но это не так уж и важно, поскольку браузер не интересует, насколько красиво написаны исходные файлы JavaScript, главное чтобы их можно было выполнить.

Если параметры компилятора TypeScript были настроены именно так, как указано в данном рецепте, то компилятор выведет файл с расширением `.js` и с тем же самым именем, расположив его рядом с исходным файлом, имеющим расширение `.ts`.

Дополнение

Действие компилятора TypeScript далеко не ограничивается формированием единичного файла с расширением `.ts`. В нем предоставляется обширный ряд инструментальных средств для указания, как должны выглядеть файлы, выводимые при компиляции.

Формирование карт исходного кода

Компилятор TypeScript способен также формировать карты исходного кода в дополнение к выходным файлам. Если вы незнакомы с ними, то имейте в виду, что употребление карт исходного кода объясняется характером компиляции и минимизации, когда файлы, отлаживаемые в браузере, не являются исходными файлами, написанными вами. Более того, исходные файлы TypeScript могут быть даже скомпилированы из файлов, написанных на другом языке.

Карты исходного кода являются индексами, сопутствующими скомпилированным файлам и предназначаемыми для описания их первоначального вида до компиляции. В частности, файлы с расширением `.js.map` содержат схему кодировки, связывающую скомпилированные и/или минимизированные маркеры с их первоначальными именами и структурами в исходном, нескомпилированном файле с расширением `.ts`. А в тех браузерах, где известно, как пользоваться картами исходного кода, можно воссоздать

первоначальный вид файла и позволить разработчиками устанавливать точки останова, пошагово выполнять отлаживаемый код и проверять лексические конструкции в нем таким же образом, как и в исходном файле.

Карты исходного кода могут быть указаны с помощью следующей специальной директивы, добавляемой в скомпилированный файл:

```
// #sourceMappingURL=/dist/example.js.map
```

Если требуется сформировать файлы карт исходного кода для их последующего вывода, это можно указать в файле конфигурации, а также добавить настройку `"sourceMap": true`. По умолчанию файлы с расширением **`.js.map`** будут создаваться в том же самом месте, где и выходные файлы с расширением **`.js`**. А с другой стороны, компилятору можно дать указание сформировать карты исходного кода в самом выходном файле с расширением **`.js`**.

Несмотря на то что дополнительные файлы карт исходного кода не оказывают никакого влияния на окончательное поведение приложения, встраивать их в выходные файлы с расширением **`.js`** нежелательно, чтобы не раздувать излишне размер этих файлов. Дело в том, что клиенты, которым совсем не нужны файлы карт исходного кода, должны будут все равно их загрузить.

Компиляция в единый файл

Все скомпонованные модули проверяются компилятором TypeScript на предмет импорта и экспорта, и поэтому все скомпилированные файлы совсем не обязательно должны однозначно соответствовать своим входным файлам. Компилятор вполне способен объединить скомпилированные файлы в единый файл, если такую возможность поддерживает формат выходных модулей. Единый файл можно указать там, где требуется скомпилировать все модули, с помощью настройки `"outFile": "/dist/bundle.js"`.

Некоторые форматы выходных модулей (например, в библиотеке CommonJS) не будут работать в виде сцепленных модулей в едином файле, и поэтому их нельзя указывать в свойстве **`outFile`**. Однако начиная с версии TypeScript 1.8, поддерживаются форматы выходных AMD- и системных модулей.

Если предполагается пользоваться библиотекой SystemJS, то данный параметр настройки компилятора может оказать определенную помощь, поскольку в библиотеке SystemJS поддерживаются практически любые форматы модулей. Но если используется сборщик, ориентированный на библиотеку CommonJS (например, Webpack), то объединение файлов лучше поручить именно ему.

См. также

- Рецепт "Содержимое файла `package.json` для минимально жизнеспособного приложения Angular 2" с описанием составных частей основного файла конфигурации для разрабатываемого проекта.
- Рецепт "Транспиляция в браузере средствами библиотеки SystemJS", в котором демонстрируется, как воспользоваться библиотекой SystemJS, чтобы связать вместе нескомпилированные статические файлы.
- Рецепт "Составление файлов приложений для минимально жизнеспособного приложения Angular 2", в котором постепенно разъясняется, как создать очень простое приложение Angular 2 с самого начала.
- Рецепт "Перенос минимально жизнеспособного приложения Angular 2 в сборку Webpack" с описанием порядка внедрения Webpack в процесс написания приложения в Angular.

Транспиляция в браузере средствами библиотеки SystemJS

Нередко оказывается полезно иметь возможность доставлять исходные файлы TypeScript непосредственно в браузер, откладывая их транспиляцию до тех пор, пока они реально не понадобятся. И хотя такой способ чреват снижением производительности, он очень удобен на этапах разработки прототипов и экспериментирования.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **`http://ngcookbook.herokuapp.com/2283/`**.

Подготовка

Создайте пустой каталог для своего проекта, а нем — файл конфигурации `package.json` с приведенным ниже содержимым. Выполнив команду **`npm install`**, вы подготовите все к написанию кода:

```
[package.json]

{
  "scripts": {
    "lite-server": "lite-server"
  },
  "devDependencies": {
    "lite-server": "^2.2.2",
    "systemjs": "^0.19.38",
    "typescript": "^2.0.3"
  }
}
```

Реализация

Пакет npm под названием TypeScript содержит встроенный транспилятор. После объединения его с библиотекой SystemJS в качестве назначаемой утилиты транспиляции у вас появляется возможность направлять исходные файлы TypeScript клиенту, после чего библиотека SystemJS будет транспилировать их в код JavaScript, совместимый с браузером.

Прежде всего создайте файл index.html, приведенный ниже. Сначала в этом файле импортируются следующие две библиотеки JavaScript: system.js и typescript.js, а затем указывается требующийся транспилятор Typescript и импортируется файл верхнего уровня main.ts:

```
[index.html]

<html>
<head>
  <script src="node_modules/systemjs/dist/system.js">
  </script>
  <script src="node_modules/typescript/lib/typescript.js">
  </script>
  <script>
    System.config({
      transpiler: 'typescript'
    });
    System.import('main.ts');
  </script>
</head>
<body>
  <h1 id="text"></h1>
</body>
</html>
```

Далее создайте файл TypeScript верхнего уровня со следующим содержимым:

```
[main.ts]

import {article} from './article.ts';

document.getElementById('text')
        .innerHTML = article;
```

И, наконец, создайте файл TypeScript с приведенной ниже зависимостью. В итоге вы сможете запустить сервер разработки по команде **npm run lite-server** и увидеть, что приложение TypeScript нормально выполняется в браузере по адресу localhost:3000.

```
[article.ts]

export const article = "Cool story, bro";
```

Принцип действия

Библиотека SystemJS позволяет разрешить зависимости в модуле, а также применить к модулю транспилятор, прежде чем он попадет в браузер. Если проанализировать транспилированные файлы в инспекторе браузера, то можно заметить, что распространяемые файлы имеют вид характерных для JavaScript *немедленно вызываемых функциональных выражений* (IIFE), а также сопутствующих им карт исходного кода. С помощью этих инструментальных средств можно написать необыкновенно сложные приложения, вообще не прибегая к управлению файлами на стороне сервера.

Дополнение

Выполнять транспиляцию в браузере стоит лишь в целях экспериментирования или разработки чернового варианта проекта. Любые вычисления следует выполнять при всякой возможности на сервере. Кроме того, все клиенты, транспилирующие свои файлы, выполняют немало избыточных операций, поскольку все они транспилируют одни и те же файлы.

См. также

- Рецепт "Содержимое файла `package.json` для минимально жизнеспособного приложения Angular 2" с описанием составных частей основного файла конфигурации для разрабатываемого проекта.
- Рецепт "Конфигурирование TypeScript для минимально жизнеспособного приложения Angular 2", в котором постепенно разъясняется, как создать очень простое приложение Angular 2 с самого начала.
- Рецепт "Составление файлов приложений для минимально жизнеспособного приложения Angular 2" с описанием порядка внедрения Webpack в процесс написания приложения в Angular.
- Рецепт "Перенос минимально жизнеспособного приложения Angular 2 в сборку Webpack", где описывается порядок внедрения Webpack в процесс написания приложения в Angular.
- Рецепт "Внедрение прокладок и полифилов в Webpack", в котором предлагается удобный способ управления зависимостями полифилов в Angular 2.
- Рецепт "Формирование HTML-разметки с помощью подключаемого модуля `html-webpack-plugin`", где показывается, как настроить пакет `npm` на автоматическое добавление скомпилированных файлов в разметку HTML-документа.

- Рецепт "Установка приложения Angular с помощью интерфейса командной строки" с описанием особенностей применения интерфейса командной строки, что он дает разработчикам и каково назначение его отдельных частей.

Составление файлов приложений для минимально жизнеспособного приложения Angular 2

Приступая к освоению Angular 2, полезно иметь ясное представление о структуре приложения, которая сводится к самым простым компонентам. В частности, минимально жизнеспособное приложение будет состоять из единственного компонента. Эта глава посвящена вопросам организации приложений, и поэтому в ней основное внимание уделяется не тому, как должен выглядеть компонент, а тому, как взять определение компонента в TypeScript и визуализировать его на веб-странице.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/6323/**.

Подготовка

В данном рецепте предполагается, что вы выполнили все инструкции, указанные ранее для составления файлов конфигурации минимально жизнеспособного приложения Angular 2. В итоге установка модуля `npm` по следующей команде должна пройти успешно и без ошибок:

```
npm install
```

Реализация

Начать проще всего с базового компонента приложения. Ниже поясняется, как это делается.

Файл `app.component.ts`

Реализуйте компонент в новом каталоге `app/`, как показано ниже. В нем нет ничего особенного. Это самый простой компонент, который только можно получить. Как только этот компонент будет успешно визуализирован на стороне клиента, на экране должна появиться единственная текстовая строка с крупным заголовком.

```
[app/app.component.ts]

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

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

Файл app.module.ts

Далее необходимо определить модуль с помощью декоратора `@NgModule`, который будет связан с данным компонентом. С этой целью создайте в каталоге `app/` еще один файл `app.module.ts` со следующим содержимым:

```
[app/app.module.ts]

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

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

Ниже дополнительно поясняется, что происходит в данном файле.

- В свойствах `import` указываются модули, директивы и каналы которых должны быть доступны в данном модуле.

Импорт модуля **`BrowserModule`** дает возможность получить доступ к базовым директивам вроде **`ngIf`** и указать тип визуализации, порядок управления событиями и тип документа. Если приложение выполняет визуализацию в веб-браузере, данный модуль предоставляет требующиеся для этой цели инструментальные средства.

- В свойстве `declarations` указываются директивы и каналы, экспортируемые в этом модуле. В данном случае экспортируется только компонент `AppComponent`.
- В свойстве `bootstrap` указываются компоненты, которые должны исходно загружаться вместе с данным модулем. В частности, перечисленные здесь компоненты предназначены для визуализации в пределах данного модуля. Так, компонент `AppComponent` может быть исходно загружен и визуализирован в каком-то другом месте. Именно там данная спецификация и возымеет действие.

На этом определение данного модуля завершается. Итак, вы успешно связали компонент с его модулем, но этот модуль пока еще никуда исходно не загружается и даже никуда не включается.

Файл `main.ts`

Измените это положение вещей, внеся следующие коррективы в исходный файл `main.ts` верхнего уровня TypeScript:

```
[app/main.ts]

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

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

В этом файле определяется декоратор `@NgModule`, предназначаемый для компонента `AppComponent`. В нем указывается, что в данный модуль необходимо импортировать модуль `BrowserModule`.

Напомним, что каркас Angular 2 разработан как платформенно-независимый. В частности, он побуждает разработчиков писать прикладной код, который совсем не обязательно может выполняться в обыкновенном веб-браузере. В данном случае целевым выбран стандартный веб-браузер, и об этом можно известить приложение, импортировав модуль **`BrowserModule`** из целевого пакета **`platform-browser-dynamic`**. Если бы требовалось специально указать целевую платформу, для этого пришлось быть выбрать другую платформу для импорта в корневой компонент приложения.

В данном объявлении декоратора `@NgModule` указывается также, что компонент `AppComponent` существует и должен быть исходно загружен.

Начальная загрузка обозначает порядок запуска приложения Angular 2, хотя она имеет весьма специфическое определение. В частности, вызов метода **`bootstrap()`** означает, что каркас Angular должен подключить указанный компонент приложения к элементам модели DOM, обозначаемым селектором данного компонента. В итоге запускается первоначальный цикл обнаружения изменений и их побочных эффектов, и на этом завершается инициализация компонента.

Из данного объявления следует, что при начальной загрузке данного модуля будет исходно загружен и компонент `AppComponent`. Следовательно, данный модуль будет, в свою очередь, исходно загружен из исходного файла TypeScript верхнего уровня. В карасе Angular 2 это условие специально задается в файле `main.ts` следующим образом:

```
[app/main.ts]

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

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

Метод `platformBrowserDynamic()` возвращает объект платформы, предоставляющий доступ к методу `bootstrapModule()`. Тем самым конфигурируется начальная загрузка приложения динамическим (JIT) компилятором Angular 2.

Причины, по которым указывается динамическая компиляция, в данный момент особого значения не имеют. Достаточно сказать, что динамическая компиляция является упрощенным вариантом по сравнению со статической компиляцией в Angular 2.

Файл `index.html`

И, наконец, необходимо создать HTML-файл, способный связать в единый комплект все скомпилированные файлы и начать инициализацию приложения. Начните его создание со следующей разметки минимально жизнеспособного приложения:

```
[index.html]

<html>
<head>
  <title>Angular 2 Minimum Viable Application</title>
  <script src="node_modules/zone.js/dist/zone.js">
  </script>
  <script src="node_modules/reflect-metadata/Reflect.js">
  </script>
  <script src="node_modules/systemjs/dist/system.src.js">
  </script>
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

Именно такую разметку в основном и следовало ожидать. В частности, ZoneJS и Reflect обозначают в Angular 2 зависимости от соответствующих библиотек, а в качестве загрузчика модулей служит библиотека SystemJS. И, наконец, элемент разметки `<app-root>` обозначает корневой компонент приложения, содержимое которого визуализируется в компоненте типа `AppComponent`.

Конфигурирование библиотеки SystemJS

Далее необходимо сконфигурировать библиотеку SystemJS таким образом, чтобы стало понятно, как импортировать файлы модулей и устанавливать связь с импортируемыми модулями в других модулях. Иными словами, в этой библиотеке необходимо задать главный файл, с которого должен начинаться данный процесс, а также каталог сопоставлений для зависимостей в этом файле. Для этой цели служат методы `System.config()` и `System.import()`, вызываемые для глобального объекта типа `System`, как выделено ниже полужирным шрифтом.

[index.html]

```
<html>
<head>
  <title>Angular 2 Minimum Viable Application</title>
  <script src="node_modules/zone.js/dist/zone.js">
  </script>
  <script src="node_modules/reflect-metadata/Reflect.js">
  </script>
  <script src="node_modules/systemjs/dist/system.src.js">
  </script>
  <script>
    System.config({
      paths: {
        'ng:': 'node_modules/@angular/'
      },
      map: {
        '@angular/core': 'ng:core/bundles/core.umd.js',
        '@angular/common': 'ng:common/bundles/common.umd.js',
        '@angular/compiler':
                'ng:compiler/bundles/compiler.umd.js',
        '@angular/platform-browser':
                'ng:platform-browser/bundles/
                   ↳platform-browser.umd.js',
        '@angular/platform-browser-dynamic':
                'ng:platform-browser-dynamic/bundles/
                    ↳platform-browserdynamic.umd.js',
        'rxjs': 'node_modules/rxjs'
      },
      packages: {
        app: {
         main: './main.js'
        },
        rxjs: {
          defaultExtension: 'js'
```

```
        }
      }
    });
    System.import('app');
  </script>
</head>

<body>
  <app-root></app-root>
</body>
</html>
```

В методе `System.config()` указывается порядок обработки файлов, передаваемых библиотеке SystemJS. В частности:

- В свойстве `paths` задается псевдоним для сокращенного обозначения пути в свойстве `map`. Оно действует как простая функция поиска и замены. Таким образом, любые обнаруженные экземпляры псевдонима `ng:` заменяются на путь `node_modules/@angular/`.
- В свойстве `map` задается порядок разрешения в библиотеке SystemJS операций импорта для не указанных явно модулей. Здесь этот порядок принимает форму пяти базовых модулей Angular и библиотеки RxJS.
- В свойстве `packages` задаются целевые пакеты для импорта, а также сопоставляемые с ними файлы.

Например, свойство **app** может быть использовано при импорте модуля `app`, и в этом случае библиотека SystemJS сопоставит его с главным файлом **main.js**. Аналогично, если в данном модуле потребуется модуль из библиотеки RxJS (например, **Subject**), библиотека последует по пути импорта **rxjs/Subject**, распознает обозначение **js** как стандартное расширение в свойстве **defaultExtension**, сопоставит модуль с представляющим его файлом **node_modules/rxjs/Subject.js** и, наконец, импортирует его.

См. также

- Рецепт "Содержимое файла `package.json` для минимально жизнеспособного приложения Angular 2" с описанием составных частей основного файла конфигурации для разрабатываемого проекта.
- Рецепт "Конфигурирование TypeScript для минимально жизнеспособного приложения Angular 2", в котором поясняется, как настроить компиляцию на поддержку проекта Angular 2.
- Рецепт "Транспиляция в браузере средствами библиотеки SystemJS", где демонстрируется, как воспользоваться библиотекой SystemJS, чтобы связать вместе нескомпилированные статические файлы.

- Рецепт "Перенос минимально жизнеспособного приложения Angular 2 в сборку Webpack" с описанием порядка внедрения Webpack в процесс написания приложения в Angular.
- Рецепт "Внедрение прокладок и полифилов в Webpack", где предлагается удобный способ управления зависимостями полифилов в Angular 2.
- Рецепт "Формирование HTML-разметки с помощью подключаемого модуля `html-webpack-plugin`", в котором показывается, как настроить пакет `npm` на автоматическое добавление скомпилированных файлов в разметку HTML-документа.
- Рецепт "Установка приложения Angular с помощью интерфейса командной строки" с описанием особенностей применения интерфейса командной строки, что он дает разработчикам и каково назначение его отдельных частей.

Перенос минимально жизнеспособного приложения Angular 2 в сборку Webpack

Имеется немало причин, по которым было бы выгодно как можно проще и быстрее загружать и выполнять на стороне клиента прикладной код, получаемый от сервера. Этого проще и эффективнее всего добиться, собрав весь прикладной код в единый файл. И почти во всех случаях было бы очень эффективно, если бы браузер загружал единый файл, содержащий все зависимости, которые требуются для начальной загрузки приложения.

В упаковщике Webpack предоставляется немало полезных инструментальных средств, к числу которых относится сборщик пакетов JavaScript. В данном рецепте демонстрируется, как объединить все файлы приложения, включая зависимости от пакетов `npm`, в единый исходный файл JavaScript, предназначаемый для обработки в браузере.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **`http://ngcookbook.herokuapp.com/3310/`**.

Подготовка

Непременно выполните все инструкции, указанные в предыдущих рецептах, для составления файлов конфигурации и файлов минимально жизнеспособного приложения Angular 2. В итоге по команде **`npm start`** должен быть запущен сервер разработки, доступный по адресу `localhost:3000`.

Реализация

Начните с удаления зависимости данного приложения от библиотеки SystemJS. Для разрешения зависимостей и сборки всех файлов в единый исходный файл JavaScript служит упаковщик Webpack. Поэтому установите сначала Webpack по следующей команде:

```
npm install webpack -g
```

Файл webpack.config.js

Упаковщик Webpack ищет в файле конфигурации `webpack.config.js` инструкции по дальнейшему поведению. Поэтому создайте этот файл со следующим содержимым:

```
[webpack.config.js]

module.exports = {
  entry: "./app/main.js",
  output: {
    path: "./dist",
    filename: "bundle.js"
  }
};
```

Как видите, в этом файле нет ничего сложного. В нем содержатся инструкции упаковщику Webpack выбрать файл `main.js` на верхнем уровне приложения, разрешить все его зависимости в тех файлах, где они определены, а также собрать их в единый файл `bundle.js`, расположив его в каталоге `dist/`.

Чтобы убедиться в работоспособности всего описанного выше на данном этапе, вызовите из командной строки упаковщик Webpack, который, в свою очередь, запустит сборщик пакетов. В итоге исходный файл **bundle.js** должен появиться в каталоге **dist/** вместе со всеми указанными в нем зависимостями от модулей.

Итак, положено совсем неплохое начало, но сформированный выше файл `bundle.js` пока еще нигде не применяется. Чтобы воспользоваться им, внесите в файл `index.html` изменения, выделенные ниже полужирным шрифтом.

```
[index.html]

<html>
<head>
  <title>Angular 2 Minimum Viable Application</title>
  <script src="node_modules/zone.js/dist/zone.js">
  </script>
  <script src="node_modules/reflect-metadata/Reflect.js">
```

```
  </script>
  <script src="dist/bundle.js">
  </script>
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

Такого вы, вероятно, совсем не ожидали! Ведь теперь файл `bundle.js` служит точкой входа в приложение, а для разрешения зависимостей от любых модулей библиотека SystemJS больше не требуется, поскольку это уже сделано автоматически при сборке файлов в Webpack. Следовательно, зависимость приложения от библиотеки SystemJS можно удалить из файла конфигурации `package.json`, заменив ее другой зависимостью от сценариев Webpack, как показано ниже.

[package.json]

```
{
  "name": "mva-bundling",
  "scripts": {
    "start": "tsc && webpack && concurrently 'npm run tsc:w'
    'npm run wp:w' 'npm run lite'",
    "lite": "lite-server",
    "postinstall": "npm install -S @types/node @types/core-js",
    "tsc": "tsc",
    "tsc:w": "tsc -w",
    "wp": "webpack",
    "wp:w": "webpack --watch"
  },
  "dependencies": {
    "@angular/common": "2.0.0",
    "@angular/compiler": "2.0.0",
    "@angular/core": "2.0.0",
    "@angular/platform-browser": "2.0.0",
    "@angular/platform-browser-dynamic": "2.0.0",
    "core-js": "^2.4.1",
    "reflect-metadata": "^0.1.3",
    "rxjs": "5.0.0-beta.12",
    "zone.js": "^0.6.23"
  },
  "devDependencies": {
    "concurrently": "^2.2.0",
    "lite-server": "^2.2.2",
    "typescript": "^2.0.2",
    "webpack": "^1.13.2"
  }
}
```

Следует ли указывать имеющиеся здесь зависимости от Webpack и Typescript в свойствах `typescript` и `webpack` из раздела `devDependencies` в файле конфигурации `package.json` — это вопрос спорный, поскольку многое зависит от того, как вы организуете управление своей локальной средой. Если вы уже установили эти средства с помощью глобального параметра, то перечислять их в качестве зависимостей уже не нужно. Дело в том, что диспетчер `npm` автоматически выполняет поиск глобально установленных пакетов перед запуском сценария `npm`. Если же указать в свойстве `webpack` упаковщик Webpack, то будет установлен его дубликат, локальный для данного проекта, что, очевидно, избыточно.

Но для целей данного рецепта это все же стоит сделать, поскольку гарантирует, что по единственной команде **`npm install`** в командной строке можно извлечь сразу все требующиеся пакеты, указав нужные их версии в самом проекте.

Если теперь выполнить команду **`npm start`**, то произойдет следующее.

- Компилятор TypeScript выполнит первоначальную компиляцию исходных файлов с расширением **`.ts`** в исходные файлы с расширением **`.js`**.
- Упаковщик Webpack выполнит первоначальную сборку всех исходных файлов JavaScript в единый файл `bundle.js` и расположит его в каталоге `dist/`.
- Одновременно запустится сервер `lite-server`, а также наблюдатели компилятора TypeScript и упаковщика Webpack. Внеся необходимые изменения в исходный файл с расширением **`.ts`**, компилятор TypeScript скомпилирует его в исходный файл с расширением **`.js`**, а упаковщик Webpack воспримет изменения в этом файле и снова упакует его в файл `bundle.js`. И, наконец, сервер `lite-server` обнаружит изменения в файле `bundle.js` и перезагрузит страницу, чтобы можно было увидеть автоматически обновленные изменения.

Если не указать более конкретную конфигурацию, компилятор TypeScript, упаковщик Webpack и сервер **`lite-server`** будут использовать стандартные настройки для наблюдения за списками файлов, которые могут оказаться слишком обширными, а следовательно, наблюдению будут подлежать излишние файлы. В идеальном случае компилятор TypeScript будет наблюдать только за исходными файлами с расширением **`.js`**, как и указано в файле конфигурации **`tsconfig.json`**; упаковщик Webpack — только за файлами с расширением **`.html`**, **`.js`** и **`.css`**; а сервер **`lite-server`** — только за теми файлами, которые обслуживаются по запросам клиента.

См. также

- Рецепт "Внедрение прокладок и полифилов в Webpack", где предлагается удобный способ управления зависимостями полифилов в Angular 2.
- Рецепт "Формирование HTML-разметки с помощью подключаемого модуля `html-webpack-plugin`", в котором показывается, как настроить пакет `npm` на автоматическое добавление скомпилированных файлов в разметку HTML-документа.

Внедрение прокладок и полифилов в Webpack

До сих пор реализация рассматриваемого здесь приложения была довольно ясной, но в файле `index.html` по-прежнему имеются две висячие прокладки. И хотя файл `index.html` упрощен до такой степени, что теперь запрашивается лишь несколько исходных файлов JavaScript вместо каждого целевого модуля по отдельности, можно пойти еще дальше и собрать все эти файлы в единый файл.

Трудность состоит в том, что браузерные прокладки (browser shims) не предоставляются через модули. Иными словами, их нельзя импортировать из отдельных файлов для последующего применения. Они только допускают свое применение. Следовательно, при стандартной сборке в Webpack эти целевые компоненты не выбираются и не включаются в собранный файл.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **`http://ngcookbook.herokuapp.com/7479/`**.

Подготовка

Прежде всего завершите все, что предписано в предыдущем рецепте "Перенос минимально жизнеспособного приложения Angular 2 в сборку Webpack", чтобы иметь все исходные файлы, требующиеся в данном рецепте.

Реализация

Решить поставленную здесь задачу можно несколькими способами, включая добавление подключаемых модулей Webpack. Но самый простой из них состоит в том, чтобы просто добавить операторы импорта вручную. С этой целью создайте новый файл `polyfills.ts` со следующим содержимым:

```
[src/polyfills.ts]

import "reflect-metadata";
import "zone.js";
```

Затем импортируйте следующий модуль из файла `main.ts`:

```
[src/main.ts]

import './polyfills';
import {platformBrowserDynamic}
        from '@angular/platform-browser-dynamic';
import {AppModule} from './app/app.module';

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

И, наконец, отредактируйте файл `index.html` следующим образом:

```
[index.html]

<html>
<head>
  <title>Angular 2 Minimum Viable Application</title>
<body>
  <app-root></app-root>
  <script src="dist/bundle.js"></script>
</body>
</html>
```

Теперь упаковщик Webpack должен быть в состоянии разрешить импорт прокладок. А требующиеся файлы будут включены в собранный файл `bundle.js`.

Принцип действия

Единственная причина, по которой полифилы не обнаруживаются в Webpack, заключается в том, что они не требуются нигде в приложении. А там, где они используются, считается, что целевые ресурсы вроде библиотеки `Zone.js` сделаны доступными ранее. Следовательно, их нетрудно импортировать на самый верхний уровень приложения, где им и место. В таком случае с помощью Webpack можно обнаружить наличие полифилов и внедрить их в сформированную сборку.

См. также

- Рецепт "Перенос минимально жизнеспособного приложения Angular 2 в сборку Webpack", где описывается порядок внедрения Webpack в процесс написания приложения в Angular.
- Рецепт "Формирование HTML-разметки с помощью подключаемого модуля `html-webpack-plugin`", в котором показывается, как настроить пакет `npm` на автоматическое добавление скомпилированных файлов в разметку HTML-документа.

Формирование HTML-разметки с помощью подключаемого модуля html-webpack-plugin

В идеальном случае было бы желательно, чтобы упаковщик Webpack управлял собранным файлом и его внедрением в шаблон. Но по умолчанию упаковщик Webpack не в состоянии этого сделать, поскольку его интересуют только файлы, связанные с созданием сценариев. Правда, в Webpack реализован очень популярный подключаемый модуль, позволяющий расширить пределы заинтересованности Webpack в файлах.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/7185/**.

Подготовка

Установите подключаемый модуль `html-webpack-plugin` по приведенной ниже команде и укажите его в разделе `devDependencies` файла конфигурации `package.json`.

```
npm install html-webpack-plugin --save-dev
```

Реализация

Прежде всего подключаемый модуль `html-webpack-plugin` необходимо добавить в файл конфигурации `package.json`, если этого еще не было сделано. Ниже показано, как это делается.

```
[package.json]

{
  "name": "mva-bundling",
  "scripts": {
    "start": "tsc&&webpack&& concurrently 'npm run tsc:w'
             'npm run wp:w' 'npm run lite'",
    "lite": "lite-server",
    "postinstall": "npm install -S @types/node @types/core-js",
    "tsc": "tsc",
    "tsc:w": "tsc -w",
    "wp": "webpack",
    "wp:w": "webpack --watch"
  },
  "dependencies": {
    "@angular/common": "2.0.0",
    "@angular/compiler": "2.0.0",
    "@angular/core": "2.0.0",
    "@angular/platform-browser": "2.0.0",
```

```
    "@angular/platform-browser-dynamic": "2.0.0",
    "core-js": "^2.4.1",
    "reflect-metadata": "^0.1.3",
    "rxjs": "5.0.0-beta.12",
    "zone.js": "^0.6.23"
  },
  "devDependencies": {
    "concurrently": "^2.2.0",
    "html-webpack-plugin": "^2.22.0",
    "imports-loader": "^0.6.5",
    "lite-server": "^2.2.2",
    "typescript": "^2.0.2",
    "webpack": "^1.13.2"
  }
}
```

Как только этот подключаемый модуль будет установлен, определите порядок его действия в файле конфигурации Webpack:

[webpack.config.js]

```
var HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
  entry: "./src/main.js",
  output: {
  path: "./dist",
  filename: "bundle.js"
},
  plugins: [new HtmlWebpackPlugin({
    template: './src/index.html'
  })]
};
```

Подобным образом указывается выходной HTML-файл, предназначенный для работы всего приложения в целом. А поскольку подключаемый модуль `html-webpack-plugin` сгенерирует этот HTML-файл автоматически, необходимо внести коррективы в уже существующий файл, предназначенный в качестве шаблона:

[src/index.html]

```
<html>
<head>
  <title>Angular 2 Minimum Viable Application</title>
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

И, наконец, необходимо сконфигурировать сервер разработки для обработки на нем файлов, поскольку файл `index.html` теперь обрабатывается из каталога `dist/`. Сервер `lite-server` служит лишь оболочкой для подключаемого модуля BrowserSync, и поэтому в свойстве `baseDir` из файла конфигурации `bs-config.json` достаточно указать базовый каталог, создав данный файл, как показано ниже.

```
[bs-config.json]

{
  "server": { "baseDir": "./dist" }
}
```

Принцип действия

Упаковщик Webpack полностью осведомлен о формируемой им сборке, и поэтому имеет смысл поддерживать ссылку на эту сборку (или несколько сборок), осуществляя непосредственную переадресацию по каналу в файл `index.html`. А подключаемый модуль `html-webpack-plugin` присоединит сценарии в конце тела HTML-документа для гарантии, что к этому моменту будет сформирована первоначальная модель DOM.

См. также

- Рецепт "Перенос минимально жизнеспособного приложения Angular 2 в сборку Webpack", где описывается порядок внедрения Webpack в процесс написания приложения в Angular.
- Рецепт "Внедрение прокладок и полифилов в Webpack", в котором предлагается удобный способ управления зависимостями полифилов в Angular 2.

Установка приложения Angular с помощью интерфейса командной строки

Вместе с каркасом Angular 2 предоставляется инструментальное средство, позволяющее создавать, собирать и выполнять приложение Angular 2 в готовом виде. Более того, в его состав входит генератор, способный формировать в режиме командной строки совместимые по стилю оформления файлы и каталоги для различных частей приложения.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/4068/**.

Подготовка

Интерфейс командной строки Angular CLI представляет собой отдельный модуль npm. Для совместимости с интерфейсом Angular CLI необходимо установить платформу Node.js, начиная с версии 7.0.0.

Можно также управлять средами Node с помощью nvm — диспетчера версий платформы Node. При этом вы получаете в свое распоряжение прозрачную оболочку, а также пакеты npm в данной среде. Если вам приходилось использовать громоздкую команду **sudo npm install -g**, вы по достоинству оцените все удобства такого инструментального средства.

Установив платформу Node.js (и выбрав подходящую среду, если вы воспользовались диспетчером nvm), установите далее интерфейс Angular CLI по следующей команде:

```
npm install -g angular-cli
```

Реализация

Интерфейс Angular CLI пригоден для формирования полностью работоспособного приложения Angular 2 в режиме командной строки. Чтобы создать приложение под названием PublisherApp, выполните следующую команду:

```
ng new publisher
```

Интерфейс Angular CLI добросовестно соберет все файлы, необходимые для формирования минимально жизнеспособного приложения Angular 2 на языке TypeScript, инициализирует информационное хранилище Git и установит все требующиеся зависимости. Список созданных в итоге файлов должен выглядеть следующим образом:

```
create README.md
createsrc/app/app.component.css
createsrc/app/app.component.html
createsrc/app/app.component.spec.ts
createsrc/app/app.component.ts
createsrc/app/app.module.ts
createsrc/app/index.ts
createsrc/app/shared/index.ts
createsrc/environments/environment.prod.ts
createsrc/environments/environment.ts
createsrc/favicon.ico
createsrc/index.html
createsrc/main.ts
createsrc/polyfills.ts
```

```
createsrc/styles.css
createsrc/test.ts
createsrc/tsconfig.json
createsrc/typings.d.ts
create angular-cli.json
create e2e/app.e2e-spec.ts
create e2e/app.po.ts
create e2e/tsconfig.json
create .gitignore
create karma.conf.js
createpackage.json
create protractor.conf.js
create tslint.json
```

Перейдите в каталог вновь созданного приложения по команде **cd publisher**. Это даст возможность выполнять все команды Angular CLI, связанные с данным проектом.

Локальное выполнение приложения

Чтобы выполнить данное приложение, запустите сервер по приведенной ниже команде. Начальная страница данного приложения окажется доступной по адресу `localhost:4200`.

```
ng serve
```

Тестирование приложения

Для модульного тестирования данного приложения выполните следующую команду:

```
ng test
```

А для сквозного тестирования данного приложения выполните такую команду:

```
ng e2e
```

Принцип действия

Ниже вкратце поясняется назначение каждого из упомянутых выше файлов, составляющих рассматриваемое здесь приложение.

Файлы конфигурации проекта

Ниже приведено краткое описание основных файлов конфигурации текущего проекта.

- **angular-cli.json**. В этом файле указывается порядок сборки и управления файлами и каталогами приложения.

- **package.json**. В этом файле конфигурации пакета npm указываются сценарии и целевые ресурсы командной строки, к которым привязываются команды Angular CLI.

Файлы конфигурации TypeScript

Ниже приведено краткое описание основных файлов конфигурации TypeScript.

- **tslint.json**. В этом файле указывается конфигурация пакета npm под названием tslint. Интерфейс Angular CLI автоматически выполняет статический анализ кода в исходных файлах с расширением **.ts** по команде **npm run lint**.
- **src/tsconfig.json**. В этом файле содержится часть спецификации TypeScript, где компилятор извещается, что это корневой каталог проекта TypeScript. Содержимое данного файла определяет порядок компиляции, что дает команде **tsc** возможность пользоваться данным каталогом как корневым для компиляции.
- **e2e/tsconfig.json**. Это файл конфигурации сквозного компилятора TypeScript.
- **src/typings.d.ts**. В этом файле содержится спецификация модуля npm под названием typings. Это дает возможность описывать порядок заключения в оболочку внешних модулей и их внедрения в компилятор TypeScript. В этом файле указывается также системное пространство имен для библиотеки SystemJS.

Файлы конфигурации тестов

Ниже приведено краткое описание основных файлов конфигурации тестов.

- **karma.conf.js**. Это файл конфигурации Karma — исполнителя тестов в данном проекте.
- **protractor.conf.js**. Это файл конфигурации Protractor — среды сквозного тестирования в данном проекте.
- **src/test.ts**. В этом файле конфигурации Karma описывается порядок запуска исполнителя тестов и местонахождение тестовых файлов во всем приложении.

Базовые файлы приложения

Ниже приведено краткое описание базовых файлов приложения.

- **src/index.html**. Это корневой файл приложения, предназначенный для выполнения всего одностраничного приложения. Скомпилированные

файлы JavaScript и прочие статические ресурсы будут автоматически добавляться в этот файл сценарием сборки.

- **src/main.ts**. Это исходный файл TypeScript верхнего уровня, предназначенный для начальной загрузки приложения вместе с его определением в модуле AppModule.
- **src/polyfills.ts**. Этот файл служит для хранения списка импортируемых модулей полифилов из файла main.ts.
- **src/styles.css**. Это файл глобального стилевого оформления разрабатываемого приложения.

Файлы конфигурации окружения

Ниже приведено краткое описание основных файлов конфигурации окружения.

- **src/environments/environment.ts**. Это устанавливаемый по умолчанию файл конфигурации окружения. Его содержимое переопределяется при указании других сред в процессе написания и тестирования разрабатываемого приложения.
- **src/environments/environment.prod.ts**. Это файл конфигурации рабочего окружения, выбираемый при указании параметра **--prod** в командной строке.

Файлы компонента AppComponent

В каждом приложении Angular 2 имеется компонент верхнего уровня AppComponent, который вызывается интерфейсом Angular CLI. Ниже приведено описание файлов компонента AppComponent.

- **src/app/app.component.ts**. В этом файле содержится определение класса базового компонента TypeScript. Именно здесь находится вся логика управления данным компонентом.
- **src/app/app.component.html и src/app/app.component.css**. Это файлы шаблонной обработки и стилевого оформления базового компонента AppComponent. Напомним, что стилевое оформление задается в объекте типа ComponentMetadata и инкапсулируется только в данном компоненте.
- **src/app/index.ts**. В этом файле хранятся сведения о модулях, доступных компилятору TypeScript в текущем каталоге. В нем должны быть указаны любые модули, экспортируемые в данный каталог и применяемые в других местах приложения.

Тестовые файлы компонента AppComponent

- **src/app/app.component.spec.ts**. В этом файле содержатся модульные тесты компонента AppComponent.
- **e2e/app.e2e-spec.ts**. В этом файле содержатся сквозные тесты компонента AppComponent.
- **e2e/app.po.ts**. В этом файле содержится определение объекта страницы, предназначенного для сквозного тестирования компонента AppComponent.

Дополнение

Анализируя кодовую базу всего проекта, всю совокупность файлов можно разделить на следующие категории.

- **Файлы, посылаемые браузеру.** К этой категории относятся нескомпилированные и неминимизированные файлы приложения, а также скомпилированные и минимизированные файлы. При разработке приложения требуется иметь возможность протестировать приложение локально с помощью нескомпилированных файлов, а также передать приложение в эксплуатацию вместе со скомпилированными и минимизированными файлами для оптимизации производительности браузера. Нескомпилированные и неминимизированные файлы собираются по сценариям сборки для объединения со скомпилированными и минимизированными файлами.
- **Файлы, предназначенные для тестирования.** Сами тестовые файлы обычно распределены по всему приложению и не компилируются. К этой категории относятся файлы конфигурации и сценарии тестирования, управляющие ходом выполнения тестов и определяющие место, где исполнители тестов могут найти тестовые файлы в каталоге текущего проекта.
- **Файлы, управляющие средой разработки.** В зависимости от конкретной установки одностраничное приложение может быть выполнено самостоятельно (без серверной кодовой базы) или создано вместе с солидной серверной кодовой базой, предоставляющей интерфейсы API и другие серверные средства. Хранилища быстрого старта (quickstart repositories) или генераторы приложений (например, интерфейс Angular CLI) обычно предоставляют разработчикам минимальный HTTP-сервер в качестве отправной точки для отправки браузеру статических ресурсов. Конкретный режим работы среды разработки может отличаться, но файлы, относящиеся к данной категории, определяют

порядок работы приложения как в локальных условиях, так и в условиях эксплуатации.

- **Файлы для компиляции приложения.** К этой категории относятся файлы, которые редактируются в вашем любимом текстовом редакторе, а не те файлы, которые отправляются в браузер при работе приложения. Сценарии сборки обычно настраиваются на объединение всех файлов приложения в как можно меньшее количество файлов минимально возможных размеров. И зачастую это означает, что браузеру предоставляется единственный скомпилированный файл JavaScript, а также скомпилированный CSS-файл. Файлы данной категории служат для сведения к минимуму кодовой базы, обычной компиляции исходного кода TypeScript в код JavaScript, выбора файлов конфигурации окружения и прочих зависящих от контекста файлов, а также организации включаемых и прочих файлов и зависимостей от модулей, чтобы сделать приложение работоспособным после компиляции. Как правило, создаваемые с их помощью файлы направляются в целевой каталог `dist`, где обычно находятся файлы, отправляемые браузеру на обработку в реальном приложении.

См. также

- Рецепт "Содержимое файла `package.json` для минимально жизнеспособного приложения Angular 2", где описываются составные части основного файла конфигурации для разрабатываемого проекта.
- Рецепт "Конфигурирование TypeScript для минимально жизнеспособного приложения Angular 2" с пояснениями, как настроить компиляцию на поддержку проекта Angular 2.
- Рецепт "Транспиляция в браузере средствами библиотеки SystemJS", в котором демонстрируется, как воспользоваться библиотекой SystemJS, чтобы связать вместе нескомпилированные статические файлы.
- Рецепт "Составление файлов приложений для минимально жизнеспособного приложения Angular 2", где постепенно разъясняется, как создать очень простое приложение Angular 2 с самого начала.
- Рецепт "Внедрение прокладок и полифилов в Webpack" с описанием удобного способа управления зависимостями полифилов в Angular 2.

Глава 9

Тестирование в Angular 2

В этой главе даются следующие рецепты.

- Создание минимально жизнеспособного набора модульных тестов средствами Karma, Jasmine и TypeScript
- Написание минимально жизнеспособного набора модульных тестов для простого компонента
- Написание минимально жизнеспособного набора сквозных тестов для простого приложения
- Модульное тестирование синхронной службы
- Модульное тестирование зависимого от службы компонента с помощью заглушек
- Модульное тестирование зависимого от службы компонента с помощью шпионов

Введение

Написание тестов можно сравнить с чисткой зубов. Этой гигиенической процедуры можно какое-то время не придерживаться, но в конечном итоге она вас настигнет.

Область тестирования наводнена противоречивыми идеологиями, избитыми формулировками и показными приемами. Более того, имеется солидный набор инструментальных средств для написания и выполнения тестов самыми разными способами, автоматизации, анализа степени покрытия или правильности тестов. Помимо всего этого, у каждого разработчика имеется свой особый порядок и стиль тестирования, а также требования к тестированию разрабатываемого приложения, которые могут отличаться от первоначальных требований на стадии выработки общего замысла.

В этой главе преследуется цель представить все служебные средства, доступные в каркасе Angular 2 для тестирования, а также некоторые стратегии

для развертывания этих служебных средств. Предлагаемые в этой главе рецепты сосредоточены в основном на модульных, а не сквозных тестах, поскольку в подавляющем большинстве надежные наборы составляются из модульных тестов.

Создание минимально жизнеспособного набора модульных тестов средствами Karma, Jasmine и TypeScript

Прежде чем приступить к рассмотрению особенностей тестирования приложений в Angular 2, очень важно представить поддерживающую среду, благодаря которой тестирование вообще оказывается возможным. Большинство официально доступных ресурсов Angular предоставляют тесты, выполняемые в инструментальных средствах Karma и Jasmine, и поэтому нет никаких особых причин отказываться от применения обоих этих средств, довольно удобных для проведения тестов. Но, принимая во внимание совершенно новую область языка TypeScript, проведение тестов с помощью этих инструментальных средств потребует некоторых дополнительных соображений.

В данном рецепте демонстрируется порядок составления очень простого набора модульных тестов. В качестве среды тестирования в нем применяются инструментальные средства Karma и Jasmine, для компиляции и поддержки модулей — компилятор TypeScript и упаковщик Webpack, а в качестве тестового браузера — PhantomJS. Ниже приведено краткое описание инструментальных средств, упоминаемых здесь и далее в этой главе.

- **Karma** — это исполнитель модульных тестов. Тесты выполняются через Karma из командной строки. Это средство способно запускать тестовый сервер, где известно, как найти тестовые файлы и передать их тестовому браузеру на обработку.
- **Jasmine** — это среда тестирования. Применяя ключевые слова вроде `it` и `describe`, не следует забывать, что они являются составной частью модульных тестов в Jasmine. Совместно с Karma это инструментальное средство организует доступ к написанным тестам и их выполнение.
- **PhantomJS** — это "обезглавленный" браузер, собранный в Webkit, где "обезглавленный" означает, что такой браузер действует без видимого пользовательского интерфейса, но в то же время строит модель DOM и имеет среду выполнения JavaScript. Для выполнения модульных тестов JavaScript требуется браузер, поскольку эти тесты предназначены для проведения в исполняющей среде браузера. Для проведения тестов в Karma поддерживается большое количество модулей, подключаемых

к браузерам, в том числе и к стандартным вроде Chrome и Firefox. Если внедрить эти подключаемые к браузерам модули, Karma запустит экземпляр браузера и выполнит в нем тесты. Для составления минимально жизнеспособного набора модульных тестов достаточно провести тестирование в "обезглавленном" браузере, который выведет результаты тестирования в командной строке. Если же тесты требуется выполнить в конкретном браузере, Karma предоставит сервер, непосредственно доступный для посещения из тестового браузера через указанный порт, например, по адресу `http://localhost:9876`.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **`http://ngcookbook.herokuapp.com/3998/`**.

Подготовка

Начните с файла `package.json` со следующим содержимым:

```
[package.json]

{}
```

Содержимое этого файла должно иметь корректный формат JSON, поскольку диспетчеру `npm` требуется синтаксически проанализировать и дополнить его.

Реализация

Начните с создания файла, который требуется протестировать. С этой целью добавьте в новый файл следующий исходный код TypeScript:

```
[src/article.ts]

export class Article {
  title:string =
    "Lab Mice Strike for Improved Working Conditions, Benefits"
}
```

Написание модульного теста

Определив класс `Article`, можете импортировать его в новый тестовый файл `article.spec.ts`, чтобы воспользоваться им.

По принятому соглашению тестовые файлы в Jasmine принято именовать, завершая расширением **`.spec.ts`**. Тестовые файлы, формируемые в интерфейсе Angular CLI, будут сосуществовать вместе с тем файлом, который они тестируют, хотя это совсем не обязательно. В конфигурации Karma можно в дальнейшем определить другое соглашение об именовании тестовых файлов.

Импортируйте сначала класс `Article`, а затем создайте в Jasmine пустой тестовый набор, вызвав функцию `describe()`, как показано ниже.

```
[src/article.spec.ts]

import {Article} from './article';

describe('Article unit tests', () => {
});
```

В функции `describe()` определяется специальный набор, включающий в себя символьную строку названия `"Article unit tests"` (Модульные тесты класса `Article`), а также анонимную функцию, содержащую данный набор. Один специальный набор может быть вложен в другой.

В теле функции **`describe()`** можно определить функции **`beforeEach()`** и **`afterEach()`**, которые выполняются до и после определения каждого модульного теста в наборе. Следовательно, для модульных тестов можно определить вложенную логику настройки, используя вложенные блоки функций **`describe()`**.

Напишите модульный тест в теле функции `describe()`, описывающей специальный набор, как выделено ниже полужирным шрифтом. Обратите внимание на то, что прикладной код и тест написаны на языке TypeScript.

```
[src/article.spec.ts]

import {Article} from './article';

describe('Article unit tests', () => {
  it('Has correct title', () => {
    let a = new Article();
    expect(a.title).toBe(
          "Lab Mice Strike for Improved Working Conditions,
          Benefits");
  });
});
```

Конфигурирование Karma и Jasmine

Установите Karma, интерфейс Karma CLI, Jasmine и подключаемый модуль Karma Jasmine с помощью следующих команд:

```
npm install karma jasmine-core karma-jasmine --save-dev
npm install karma-cli -g
```

Можно использовать краткий эквивалент этих команд:

```
npm i -D karma jasmine-core karma-jasmine
npm i karma-cli -g
```

Karma читает свою конфигурацию из файла `karma.conf.js`, поэтому создайте этот файл конфигурации со следующим содержимым:

```
[karma.conf.js]

module.exports = function(config) {
  config.set({
  })
}
```

Karma требуется знать, как найти тестовые файлы и пользоваться Jasmine. Поэтому добавьте в файл конфигурации `karma.conf.js` настройки, выделенные ниже полужирным шрифтом.

```
[karma.conf.js]

module.exports = function(config) {
  config.set({
    frameworks: [
      'jasmine'
    ],
    files: [
      'src/*.spec.js'
    ],
    plugins : [
      'karma-jasmine',
    ]
})
}
```

Конфигурирование PhantomJS

PhantomJS позволяет проводить тесты непосредственно из командной строки. Но для этого Karma нужно знать, как пользоваться PhantomJS. Поэтому установите сначала подключаемый модуль PhantomJS по следующей команде:

```
npm install karma-phantomjs-launcher --save-dev
```

Далее внедрите этот подключаемый модуль в конфигурацию Karma, как выделено ниже полужирным шрифтом. В итоге Karma станет известно, как выполнять тесты в PhantomJS.

```
[karma.conf.js]

module.exports = function(config) {
  config.set({
    browsers: [
      'PhantomJS'
```

```
    ],
    frameworks: [
      'jasmine'
    ],
    files: [
      'src/*.spec.js'
    ],
    plugins : [
      'karma-jasmine',
      'karma-phantomjs-launcher'
    ]
  })
}
```

Компиляция файлов и тестов в TypeScript

Проанализировав внимательно содержимое приведенного выше файла конфигурации Karma, можно заметить, что тестовые файлы, на которые делаются ссылки, на самом деле отсутствуют. А поскольку для тестирования применяется язык TypeScript, то именно на нем необходимо создать эти файлы. С этой целью установите определения типов TypeScript и Jasmine по следующей команде:

```
npm install typescript @types/jasmine --save-dev
```

Затем добавьте выделенные ниже полужирным шрифтом определения сценариев в файл `package.json`.

[package.json]

```
{
  "scripts": {
    "tsc": "tsc",
    "tsc:w": "tsc -w"
  },
  "devDependencies": {
    "@types/jasmine": "^2.5.35",
    "jasmine-core": "^2.5.2",
    "karma": "^1.3.0",
    "karma-cli": "^1.0.1",
    "karma-jasmine": "^1.0.2",
  "karma-phantomjs-launcher": "^1.0.2",
    "typescript": "^2.0.3"
  }
}
```

Создайте файл конфигурации `tsconfig.json`. В данном случае скомпилированные файлы достаточно расположить в том же самом каталоге, и для этой цели вполне подойдет файл конфигурации `tsconfig.json` со следующим простым содержимым:

```
[tsconfig.json]

{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "moduleResolution": "node"
  }
}
```

Такая конфигурация вряд ли подойдет для рабочего приложения, но она вполне пригодна для минимально жизнеспособного его варианта. В рабочем приложении скомпилированные файлы, вероятнее всего, размещаются в совершенно другом каталоге, зачастую обозначаемом как **dist/**.

Внедрение Webpack в Karma

Безусловно, требуется каким-то образом различать определения модулей для прикладного кода и его тестов. Но Karma не в состоянии сделать это самостоятельно, а следовательно, данную задачу придется решать какими-то другими средствами. И для этой цели идеально подходит Webpack. Для предварительной обработки тестовых файлов перед их передачей браузеру имеется модуль Webpack, подключаемый к Karma по следующей команде:

```
[karma.conf.js]

module.exports = function(config) {
  config.set({
    browsers: [
      'PhantomJS'
    ],
    frameworks: [
      'jasmine'
    ],
    files: [
      'src/*.spec.js'
    ],
    plugins : [
      'karma-webpack',
      'karma-jasmine',
      'karma-phantomjs-launcher'
    ],
    preprocessors: {
      'src/*.spec.js': ['webpack']
    }
  })
}
```

Написание тестового сценария

Запустить сервер Karma можно по следующей команде:

```
karma start karma.conf.js
```

По этой команде инициализируется тестовый сервер, выполняются тесты, наблюдаются изменения и снова проводятся тесты. Но это является отступлением от того правила, что исходные файлы TypeScript требуется компилировать в файлы, наблюдаемые в Karma. У компилятора TypeScript также имеется наблюдатель за файлами, оперативно перекомпилирующий их. Было бы желательно иметь оба эти средства, чтобы перекомпилировать файлы всякий раз, когда в исходный код вносятся изменения, и поэтому их целесообразно выполнять одновременно. Для решения этой задачи и предназначен пакет `concurrently`.

Пакет **concurrently** позволяет не только выполнить, но и завершить выполнение сразу нескольких команд. Ведь без этого сигнал завершения, посланный из командной строки, будет нацелен только на тот процесс, который выполнялся самым последним, а процесс, выполняющийся в фоновом режиме, будет проигнорирован.

Установите пакет `concurrently` по следующей команде:

```
npm install concurrently --save-dev
```

И, наконец, напишите тестовый сценарий для одновременного выполнения Karma и компилятора TypeScript:

```
[package.json]

{
  "scripts": {
    "test": "concurrently 'npm run tsc:w' 'karma start
            karma.conf.js'",
    "tsc": "tsc",
    "tsc:w": "tsc -w"
  },
  "devDependencies": {
    "@types/jasmine": "^2.5.35",
    "concurrently": "^3.1.0",
    "jasmine-core": "^2.5.2",
    "karma": "^1.3.0",
    "karma-cli": "^1.0.1",
    "karma-jasmine": "^1.0.2",
    "karma-phantomjs-launcher": "^1.0.2",
    "karma-webpack": "^1.8.0",
    "typescript": "^2.0.3",
    "webpack": "^1.13.2"
  }
}
```

Таким образом, вы сможете выполнять свои тесты по следующей команде:

```
npm test
```

Если все сделать правильно, то сервер Karma должен исходно загрузиться и выполнить тесты, выведя следующее сообщение:

```
PhantomJS 2.1.1 (Linux 0.0.0): Executed 1 of 1 SUCCESS
                (0.038 secs / 0.001 secs)
```

Принцип действия

Karma и Jasmine совместно направляют тестовые файлы в тестовый браузер. А в задачу компилятора TypeScript и упаковщика Webpack входит преобразование исходных файлов TypeScript в формат JavaScript, пригодный для их применения в тестовом браузере.

Дополнение

Любопытно отметить, каким образом в данном рецепте обрабатываются файлы, содержащие прикладной код и тесты. Такие файлы написаны на языке TypeScript, что дает возможность пользоваться формой записи модулей, принятой в стандарте ES6 языка JavaScript, в отличие от какой-нибудь другой смешанной стратегии. Но и в этом случае еще остаются некоторые варианты выбора режима работы установленной тестовой среды.

Разные части приложения должны проверяться в тестах поочередно, тогда как в стандартно организованном приложении действуют сразу все составляющие его модули. В данном рецепте исходные файлы TypeScript с расширением `.ts` компилируются отдельно, а получающиеся в итоге исходные файлы с расширением `.js` передаются затем для наблюдения в Karma. Такой подход, вероятно, легче усвоить тем, кто постепенно овладевает тестами, но в целом его нельзя считать самым эффективным. В Karma поддерживаются подключаемые модули TypeScript, допускающие предварительную обработку файлов TypeScript, прежде чем передать их препроцессору Webpack, а также допускается связывание отдельных стадий предварительной обработки в цепочку, что удобно для оперативной компиляции кода TypeScript по ходу предварительной обработки.

См. также

- Рецепт "Написание минимально жизнеспособного набора модульных тестов для простого компонента" с элементарным примером модульного тестирования компонентов Angular 2.
- Рецепт "Модульное тестирование синхронной службы", где демонстрируется порядок имитации внедрения зависимостей в модульных тестах.

- Рецепт "Модульное тестирование зависимого от службы компонента с помощью заглушек", в котором демонстрируется, как сымитировать службу для написания модульных тестов, избегая прямых зависимостей.
- Рецепт "Модульное тестирование зависимого от службы компонента с помощью заглушек", где показывается, как отслеживать вызовы служебных методов в модульном тесте.

Написание минимально жизнеспособного набора модульных тестов для простого компонента

Модульные тесты составляют саму основу процесса тестирования приложений. Они всегда сопутствуют исходному коду и чаще всего применяются для тестирования приложений. Они просто составляются, быстро выполняются, легко усваиваются и анализируются и способны задавать контекст, порядок применения и поведение прикладного кода.

Настройка взаимосвязанных конфигураций Karma, Jasmine, TypeScript и Angular 2 — совсем не простая задача, хотя ее рассмотрение выходит за рамки этой главы. В самом деле, было бы не очень интересно обсуждать организацию совместной работы всех этих инструментальных средств, особенно если имеется немало удачных примеров проектов, где их взаимосвязанные конфигурации уже настроены. Намного интереснее обратиться непосредственно к самим тестам, чтобы выяснить, каким образом они взаимодействуют с Angular 2.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/3935/**.

Подготовка

В данном рецепте предполагается, что вы уже пользуетесь рабочей средой тестирования в Angular 2. Для целей данного рецепта идеально подходит среда, предоставляемая в приложении, сформированном средствами интерфейса Angular CLI. В этой среде можно выполнять тесты из каталога текущего проекта с помощью следующей команды:

```
ng test
```

Итак, начните с приведенного ниже компонента. Цель состоит в том, чтобы полностью реализовать файл `article.component.spec.ts` для тестирования данного компонента.

```
[src/app/article/article.component.ts]

import {Component} from '@angular/core';
@Component({
  selector: 'app-article',
  template: `
    <h1>
      {{ title }}
    </h1>
  `
})
export class ArticleComponent {
  title: string = 'Captain Hook Sues Over Spork Snafu';
}
```

Реализация

Самый простой тест, который можно только придумать, состоит в том, чтобы проверить возможность получить экземпляр типа `ArticleComponent`. Начните со следующего теста:

```
[src/app/article/article.component.spec.ts]

import {ArticleComponent} from './article.component';

describe('Component: Article', () => {
  it('should create an instance', () => {
    let component = new ArticleComponent();
     expect(component).toBeTruthy();
  });
});
```

В этом тесте не происходит ничего особенного. А поскольку `Article Component` — это самый обычный класс TypeScript, то ничего не мешает получить его экземпляр и проверить его состояние в оперативной памяти. Но для того чтобы он вел себя как компонент Angular 2, потребуется ряд других инструментальных средств.

Применение класса `TestBed` и функции `async()`

Пытаясь приспособить среду Angular 2 к тестируемому компоненту, необходимо принять во внимание целый ряд соображений. Прежде всего, следует иметь в виду, что модульные тесты в Angular 2 сильно зависят от класса `TestBed`, который можно рассматривать как многофункциональное инструментальное средство тестирования.

Проведение модульных тестов компонента подразумевает также участие класса `ComponentFixture`. При вызове метода `TestBed.createComponent()` создается тестовая конфигурация, охватывающая экземпляр проверяемого компонента.

Потребность в тестовых конфигурациях сосредоточена на том, как должны действовать тесты. Получение экземпляра типа **ArticleComponent** не имеет особого смысла, как это было сделано в приведенном выше тесте. Здесь отсутствует присоединяемый элемент модели DOM, выполняемое приложение и т.д. В модульных тестах компонента не имеет смысла и наличие явной зависимости от подобных элементов. Таким образом, класс **ComponentFixture** служит в Angular средством, позволяющим тестировать только функциональные возможности компонента в его нормальном режиме работы, не особенно беспокоясь обо всех хлопотах, связанных с внутренне присущими ему зависимостями.

Асинхронное поведение тестовой конфигурации типа `TestBed` диктует, что тестовая логика должна выполняться в оболочке функции `async()`.

Функция **async()** просто выполняет тест в своей оболочке. Это дает исполнителю тестов возможность подождать завершения всех асинхронных вызовов в тесте до завершения самого теста.

Начните с импорта класса `TestBed` и функции `async()` из модуля тестирования Angular и добавьте заготовки еще двух модульных тестов, как выделено ниже полужирным шрифтом.

```
[src/app/article/article.component.spec.ts]

import {TestBed, async} from '@angular/core/testing';
import {ArticleComponent} from './article.component';

describe('Component: Article', () => {
  it('should create an instance', () => {
    let component = new ArticleComponent();
    expect(component).toBeTruthy();
  });

  it('should have correct title', async(() => {
  }));

  it('should render title in an h1 tag', async(() => {
  }));
});
```

Теперь у вас имеются заготовки двух тестов, которые требуется написать, и самое время воспользоваться классом `TestBed`, чтобы определить тестовый модуль. Компоненты Angular 2 сочетаются вместе с определением модуля, но при выполнении модульных тестов придется воспользоваться определением модуля в классе `TestBed` для нормальной работы компонента. И это можно сделать с помощью метода `TestBed.configureTestModule()`, который требуется вызывать перед каждым тестом.

Доступная в Jasmine функция `describe()` позволяет сгруппировать в своем теле функции `beforeEach()` и `afterEach()`, поэтому она идеально подходит в данном случае, как выделено ниже полужирным шрифтом.

```
[src/app/article/article.component.spec.ts]

import {TestBed, async} from '@angular/core/testing';
import {ArticleComponent} from './article.component';

describe('Component: Article', () => {
  it('should create an instance', () => {
    let component = new ArticleComponent();
    expect(component).toBeTruthy();
  });

  describe('Async', () => {
    beforeEach( () => {
      TestBed.configureTestingModule({
        declarations: [
          ArticleComponent
        ],
      });
    });

    it('should have correct title', async(() => {
    }));

    it('should render title in an h1 tag', async(() => {
    }));
  });
});
```

Создание тестовой конфигурации средствами класса `ComponentFixture`

Класс `TestBed` дает возможность создать тестовую конфигурацию, но это еще нужно сделать. Тестовая конфигурация потребуется для обоих тестов в функции `async()`, поэтому ее целесообразно создать и в функции `beforeEach()`:

```
[src/app/article/article.component.spec.ts]

import {TestBed, async} from '@angular/core/testing';
import {ArticleComponent} from './article.component';

describe('Component: Article', () => {
  let fixture;

  it('should create an instance', () => {
    let component = new ArticleComponent();
```

```
    expect(component).toBeTruthy();
  });

  describe('Async', () => {
    beforeEach( () => {
      TestBed.configureTestingModule({
        declarations: [
          ArticleComponent
        ],
      });

      fixture = TestBed.createComponent(ArticleComponent);
    }));

    afterEach(() => {
     fixture = undefined;
    });

    it('should have correct title', async(() => {
    }));

    it('should render title in an h1 tag', async(() => {
    }));
  });
});
```

В функции **afterEach()** переменной **fixture** присваивается неопределенное значение **undefined**. Для рассматриваемых здесь тестов это формальное излишество, но ради большей надежности рекомендуется выработать в себе привычку освобождать переменные, совместно используемые в модульных тестах. Дело в том, что самое неприятное для отладки в тестовом наборе — это растекание тестовых переменных. Ведь это всего лишь функции, последовательно выполняемые в браузере.

Определив каждый тест, можно воспользоваться его методами для инспекции получаемого экземпляра компонента самыми разными способами. В первом тесте было бы желательно проинспектировать сам объект типа `ArticleComponent` в тестовой конфигурации типа `ComponentFixture`. И это выявляется с помощью свойства `componentInstance`, как показано ниже.

```
[src/app/article/article.component.spec.ts]

import {TestBed, async} from '@angular/core/testing';
import {ArticleComponent} from './article.component';

describe('Component: Article', () => {
  let expectedTitle = 'Captain Hook Sues Over Spork Snafu';
  let fixture;

  it('should create an instance', () => {
    let component = new ArticleComponent();
```

```
    expect(component).toBeTruthy();
  });

  describe('Async', () => {
    beforeEach(async(() => {
      TestBed.configureTestingModule({
        declarations: [
          ArticleComponent
        ],
      });
      fixture = TestBed.createComponent(ArticleComponent);
    }));

    afterEach(() => {
      fixture = undefined;
    });

    it('should have correct title', async(() => {
      expect(fixture.componentInstance.title)
            .toEqual(expectedTitle);
    }));

    it('should render title in an h1 tag', async(() => {
    }));
  });
});
```

Во втором тесте требуется доступ к модели DOM, к которой присоединен экземпляр компонента в тестовой конфигурации. Корневой элемент, для которого предназначается данный компонент, выявляется с помощью свойства `nativeElement`, как показано ниже.

[src/app/article/article.component.spec.ts]

```
import {TestBed, async} from '@angular/core/testing';
import {ArticleComponent} from './article.component';

describe('Component: Article', () => {
  let expectedTitle = 'Captain Hook Sues Over Spork Snafu';
  let fixture;

  it('should create an instance', () => {
    let component = new ArticleComponent();
    expect(component).toBeTruthy();
  });

  describe('Async', () => {
    beforeEach(async(() => {
      TestBed.configureTestingModule({
        declarations: [
          ArticleComponent
        ],
      });
```

```
      fixture = TestBed.createComponent(ArticleComponent);
    }));

    afterEach(() => {
      fixture = undefined;
    });

    it('should have correct title', async(() => {
      expect(fixture.componentInstance.title)
            .toEqual(expectedTitle);
    }));

    it('should render title in an h1 tag', async(() => {
      expect(fixture.nativeElement.querySelector('h1')
            .textContent).toContain(expectedTitle);
    }));
  });
});
```

Если вы выполните эти тесты, то заметите, что последний тест не пройдет. В этом тесте проверяется пустая символьная строка в дескрипторе разметки <h1></h1>. Дело в том, что значение привязывается в шаблоне к члену компонента. А поскольку тестовая конфигурация определяет всю среду, окружающую данный компонент, то она определяет также стратегию обнаружения изменений, которая здесь не приводится в действие до тех пор, пока на то не будет дана соответствующая команда. Цикл обнаружения изменений можно начать, вызвав метод `detectChanges()`, как выделено ниже полужирным шрифтом. В итоге вы должны увидеть, как выполняется Karma и проходят все три теста.

[src/app/article/article.component.spec.ts]

```
import {TestBed, async} from '@angular/core/testing';
import {ArticleComponent} from './article.component';

describe('Component: Article', () => {
  let expectedTitle = 'Captain Hook Sues Over Spork Snafu';
  let fixture;

  it('should create an instance', () => {
    let component = new ArticleComponent();
    expect(component).toBeTruthy();
  });

  describe('Async', () => {
    beforeEach(async(() => {
      TestBed.configureTestingModule({
        declarations: [
          ArticleComponent
        ],
      });
```

```
      fixture = TestBed.createComponent(ArticleComponent);
    }));

    afterEach(() => {
      fixture = undefined;
    });

    it('should have correct title', async(() => {
      expect(fixture.componentInstance.title)
            .toEqual(expectedTitle);
    }));

    it('should render title in an h1 tag', async(() => {
      fixture.detectChanges();
      expect(fixture.nativeElement.querySelector('h1')
            .textContent).toContain(expectedTitle);
    }));
  });
});
```

Принцип действия

Если требуется протестировать компоненты, то лучшего помощника в этом деле, чем тестовая конфигурация, не найти. Она дает возможность инспектировать компоненты и манипулировать ими в удобной для их поведения среде. Кроме того, она дает возможность манипулировать экземплярами входных данных, получаемых для компонентов, а также проверять их выходные данные и результирующее поведение.

Базовый принцип модульного тестирования гласит: "нечто" тестируемое (в данном случае — класс компонента) следует интерпретировать как "черный ящик". Вы контролируете все, что направляется в "черный ящик", а ваши тесты должны выявить и определить, что именно предполагается получить на выходе из "черного ящика". Если в ваших тестах приняты во внимание все возможные варианты ввода и вывода данных, значит, вы достигли 100%-ного покрытия модульными тестами этого "черного ящика".

См. также

- Рецепт "Создание минимально жизнеспособного набора модульных тестов средствами Karma, Jasmine и TypeScript" с введением в модульные тесты на языке TypeScript.
- Рецепт "Модульное тестирование синхронной службы", где демонстрируется порядок имитации внедрения зависимостей в модульных тестах.
- Рецепт "Модульное тестирование зависимого от службы компонента с помощью заглушек", в котором демонстрируется, как сымитировать службу для написания модульных тестов, избегая прямых зависимостей.

- Рецепт "Модульное тестирование зависимого от службы компонента с помощью шпионов", где показывается, как отслеживать вызовы служебных методов в модульном тесте.

Написание минимально жизнеспособного набора сквозных тестов для простого приложения

Сквозное (end-to-end) тестирование, сокращенно обозначаемое как *e2e*, совершенно противоположно модульному тестированию. В данном случае все приложение существует в виде "черного ящика", а под вашим контролем (для проведения тестов) находятся действия, которые пользователь может предпринять в браузере, — например, инициировать события от щелчков кнопками мыши или навигации по странице. Аналогично в правильности тестов можно убедиться, лишь проверив состояние браузера и самой модели DOM.

Точнее говоря, сквозной тест запустит (в некоторой форме) конкретный экземпляр приложения (или его части), перейдет к нему в конкретном браузере, сделает свое дело на странице и проследит за тем, что на ней происходит. Это все равно, что попросить кого-нибудь сесть за компьютер и воспользоваться вашим приложением.

В данном рецепте вам предстоит составить самый элементарный набор сквозных тестов, чтобы стали более понятными основные употребляемые понятия.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/8985/**.

Подготовка

Начните с исходных файлов минимально жизнеспособного приложения, созданных ранее в рецепте из главы 8. Самыми важными файлами, которые здесь предстоит отредактировать, являются файлы `app.component.ts` и `package.json`:

```
[package.json]

{
  "scripts": {
    "start": "tsc && concurrently 'npm run tsc:w' 'npm run lite'",
    "lite": "lite-server",
    "postinstall": "npm install -s @types/node @types/core-js",
    "tsc": "tsc",
```

```
    "tsc:w": "tsc -w"
  },
  "dependencies": {
    "@angular/common": "2.1.0",
    "@angular/compiler": "2.1.0",
    "@angular/core": "2.1.0",
    "@angular/platform-browser": "2.1.0",
    "@angular/platform-browser-dynamic": "2.1.0",
    "core-js": "^2.4.1",
    "reflect-metadata": "^0.1.3",
    "rxjs": "5.0.0-beta.12",
    "systemjs": "0.19.27",
    "zone.js": "^0.6.23"
  },
  "devDependencies": {
    "concurrently": "^2.2.0",
    "lite-server": "^2.2.2",
    "typescript": "^2.0.2"
  }
}
```

```
[app/app.component.ts]

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

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

Реализация

Команда разработчиков Angular поддерживает проект каркаса Protractor, который во многих отношениях является самым лучшим средством для выполнения сквозных тестов в разрабатываемых приложениях — по крайней мере, первоначально. В состав каркаса Protractor входит огромное количество готовых утилит для манипулирования браузером при написании тестов и явных интеграций с Angular 2. Следовательно, данный проект служит отличной отправной точкой для сквозного тестирования.

Приведение Protractor в действие

Для автоматизации браузера в каркасе Protractor используется инструментальное средство Selenium. Особенности работы Selenium не особенно важны для целей создания минимально жизнеспособного набора сквозных тестов, но для этой цели необходимо установить среду выполнения Java (JRE) по следующей команде:

```
sudo apt-get install openjdk-8-jre
```

Я лично пользуюсь ОС Ubuntu, и поэтому для моих целей подходит среда выполнения OpenJDK Java Runtime Environment V8. У вас может быть другая среда разработки. Среды выполнения для разных операционных систем можно найти на веб-сайте компании Oracle.

Сам каркас Protractor может быть установлен из диспетчера `npm`, но это должно быть сделано глобально. Каркас Protractor будет использоваться вместе с Jasmine, поэтому установите его пакет вместе со средствами типизации TypeScript по следующей команде:

```
npm install jasmine-core @types/jasmine --save-dev
npm install protractor -g
```

Рассматриваемая здесь конфигурация требует дополнительного уточнения. Иногда она может оказаться вполне пригодной, если пакет **`protractor`** установлен локально, а не глобально. Ошибки, возникающие в инструментальном средстве **`webdriver-manager`**, относятся к пакету **`protractor`**, и поэтому они могут также возникнуть в связи с местом, выбранным для установки данного пакета.

Нет ничего удивительного в том, что пакет `protractor` конфигурируется в отдельном файле, поэтому создайте его со следующим содержимым:

```
[protractor.conf.js]

exports.config = {
  specs: [
    './e2e/**/*.e2e-spec.ts'
  ],
  capabilities: {
    'browserName': 'chrome'
  },
  baseUrl: 'http://localhost:3000/',
  framework: 'jasmine',
}
```

В приведенных выше настройках нет ничего особенного, как поясняется ниже.

- Файлы сквозных тестов должны находиться в каталоге `e2e/`, поэтому их имена должны оканчиваться суффиксом **`.e2e-spec.ts`**.
- Protractor должен запустить экземпляр браузера Chrome, которым можно манипулировать с помощью Selenium.
- Сервер, который требуется запустить, должен отвечать по адресу `localhost:3000`, относительно которого указываются все URL в тестах Protractor.
- Тесты Protractor должны быть написаны с помощью синтаксиса Jasmine.

Ради простоты для проведения сквозных тестов будет запущен применявшийся ранее сервер `lite-server`. После запуска сервер `lite-server` откроет собственное окно браузера, которое выглядит не совсем приглядно. А поскольку оно служит тонкой оболочкой для подключаемого модуля BrowserSync, данный сервер можно настроить, чтобы он просто не делал этого, внеся соответствующие коррективы в файл конфигурации, применяемый только для выполнения сквозных тестов.

Создайте теперь в тестовом каталоге этот файл со следующим содержимым:

```
[e2e/bs-config.json]

{
  "open": false
}
```

Этот файл не будет обнаружен в оболочке сервера **`lite-server`** автоматически. Впрочем, это положение будет исправлено ниже.

Обеспечение совместимости Protractor с Jasmine и TypeScript

Прежде всего создайте в тестовом каталоге файл конфигурации `tsconfig.json` со следующим содержимым:

```
[e2e/tsconfig.json]

{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "moduleResolution": "node"
  }
}
```

Далее создайте приведенный ниже файл с заготовкой конкретного набора сквозных тестов. Для объявления специального набора с пустым тестом здесь применяется стандартный синтаксис Jasmine.

```
[e2e/app.e2e-spec.ts]

describe('App E2E Test Suite', () => {
  it('should have the correct h1 text', () => {
  });
});
```

Прежде чем реализовать сквозной тест, необходимо добиться, чтобы файл с этим тестом фактически использовался в Protractor. Установите

сначала подключаемый модуль ts-node по приведенной ниже команде, чтобы Protractor мог компилировать и пользоваться подобным файлами в сквозных тестах.

```
npm install ts-node --save-dev
```

Затем дайте Protractor команду воспользоваться подключаемым модулем ts-node для компиляции исходных тестовых файлов в пригодном для применения формате. Это можно сделать в файле конфигурации Protractor так, как выделено ниже полужирным шрифтом. В итоге в вашем распоряжении окажется рабочий, хотя и пустой сквозной тест.

```
[protractor.conf.js]

exports.config = {
  specs: [
    './e2e/**/*.e2e-spec.ts'
  ],
  capabilities: {
    'browserName': 'chrome'
  },
  baseUrl: 'http://localhost:3000/',
  framework: 'jasmine',
  beforeLaunch: function() {
    require('ts-node').register({
      project: 'e2e'
    });
  }
}
```

Создание объекта страницы

В качестве превосходного условия для сквозного тестирования настоятельно рекомендуется пользоваться объектом страницы.

Общий замысел состоит в том, чтобы выделить всю логику взаимодействия со страницей в ее собственный объект соответствующего класса и пользоваться этим выделенным объектом страницы в другом классе для реализации тестового поведения. Это дает возможность писать тесты независимо от структуры модели DOM или определений маршрутов, облегчая тем самым сопровождение тестов. И благодаря этому тесты становятся полностью независимыми от Protractor, что упрощает дело, когда требуется сменить исполнителя сквозных тестов.

Для рассматриваемого здесь простого сквозного теста требуется указать порядок перехода на данную страницу и ее проверки, чтобы добиться

желаемого результата. С этой целью определите объект страницы с двумя компонентными методами следующим образом:

```
[e2e/app.po.ts]

import {browser, element, by} from 'protractor';

export class AppPage {
  navigate() {
    browser.get('/');
  }
  getHeaderText() {
    return element(by.css('app-root h1')).getText();
  }
}
```

Метод `navigate()` направляет Selenium по корневому пути, который, напомним, располагается по адресу `localhost:3000`. А метод `getHeaderText()` проверяет текстовое содержимое элемента модели DOM.

Следует иметь в виду, что **`browser`**, **`element`** и **`by`** — это служебные компоненты, импортируемые из пакета **`protractor`**. Подробнее о них речь пойдет далее в данном рецепте.

Написание сквозного теста

Организовав всю рассмотренную выше тестовую среду, можно без особого труда написать сквозной тест. Для каждого теста требуется получить экземпляр объекта новой страницы, как показано ниже.

```
[e2e/app.e2e-spec.ts]

import {AppPage} from './app.po';

describe('App E2E Test Suite', () => {
  let page:AppPage;

  beforeEach(() => {
    page = new AppPage();
  });

  it('should have the correct h1 text', () => {
    page.navigate();

    expect(page.getHeaderText())
          .toEqual('AppComponent template!');
  });
});
```

Написание сценариев для сквозных тестов

И, наконец, необходимо обеспечить возможность легко выполнять набор сквозных тестов. Selenium обновляется довольно часто, и поэтому данное инструментальное средство надлежит обновить явным образом перед запуском тестов.

[package.json]

```
{
  "scripts": {
    "pree2e": "webdriver-manager update && tsc",
    "e2e": "concurrently 'npm run lite -- -c=e2e/bs-config.json'
    'protractor protractor.conf.js'",
    "start": "tsc && concurrently 'npm run tsc:w' 'npm run lite'",
    "lite": "lite-server",
    "postinstall": "npm install -s @types/node @types/core-js",
    "tsc": "tsc",
    "tsc:w": "tsc -w"
  },
  "dependencies": {
    ...
  },
  "devDependencies": {
    ...
  }
}
```

И, наконец, необходимо интегрировать Angular 2 с Protractor и обеспечить возможность сообщать, когда страница готова для взаимодействия. Для этого требуется внести дополнительные коррективы в конфигурацию Protractor:

[protractor.conf.js]

```
exports.config = {
  specs: [
    './e2e/**/*.e2e-spec.ts'
  ],
  capabilities: {
    'browserName': 'chrome'
  },
  baseUrl: 'http://localhost:3000/',
    framework: 'jasmine',
    useAllAngular2AppRoots: true,
    beforeLaunch: function() {
      require('ts-node').register({
      project: 'e2e'
    });
  }
}
```

Вот, собственно, и все! Теперь вы можете выполнить набор сквозных тестов, вызвав его из соответствующего сценария npm по следующей команде:

```
npm run e2e
```

В итоге запустится экземпляр сервера lite-server (без запуска выбираемого им по умолчанию браузера), а Protractor выполнит тесты и завершит свою работу.

Принцип действия

В самом начале файла app.po.ts, где определяется объект страницы, из Protractor импортируются следующие три целевых компонента: browser, element и by. Ниже приведено их краткое описание.

- **browser** — это глобальный объект из пакета protractor, позволяющий выполнять на уровне браузера такие действия, как посещение веб-страниц по заданным URL, ожидание наступления определенных событий и получение моментальных снимков экрана.
- **element** — это глобальная функция, которой передается объект типа Locator, а она возвращает объект типа ElementFinder, который служит точкой соприкосновения для взаимодействия с соответствующим элементом модели DOM, если он существует.
- **by** — это глобальный объект, предоставляющий доступ к нескольким фабричным методам из класса Locator, предназначенным для обнаружения элементов в модели DOM. В данном случае фабричный метод by.css() выполняет действие, аналогичное вызываемому методу document.querySelector().

Полное описание программного интерфейса Protractor API можно найти по адресу: **http://www.protractortest.org/#/api**.

Причина для написания тестов подобным образом может показаться вам не совсем ясной. Ведь реальное приложение выполняется в реальном браузере, и поэтому было бы целесообразно обратиться к методам из модели DOM и аналогичным средствам.

Причина, по которой вместо этого применяется интерфейс Protractor API, довольно проста: написанный вами тестовый код не запускается непосредственно в исполняющей среде браузера. Вместо этого Protractor передает инструкции из тестового кода инструментальному средству Selenium, которое, в свою очередь, выполняет их в браузере и возвращает результаты. Таким образом, написанный вами тестовый код лишь косвенно взаимодействует с браузером и моделью DOM.

Дополнение

Назначение данного рецепта состояло в том, чтобы сформировать очень простой набор сквозных тестов и тем самым дать некоторое представление о том, что при этом происходит подспудно. И хотя сами тесты будут выглядеть более или менее в плане того, что они делают, причем независимо от тестовой среды, где они выполняются, сама тестовая среда далека от совершенства. Чтобы сделать ее более надежной, придется внести целый ряд изменений и дополнений.

При выполнении модульных тестов нередко оказывается полезно, чтобы они обнаруживали изменения в проверяемых файлах и сразу же выполняли их снова. Это объясняется, главным образом, тем, что модульные тесты должны быть сильно облегчены. Любые зависимости от остальной части приложения имитируются и абстрагируются, чтобы выполнить минимальный объем кода для подготовки среды модульного тестирования. Таким образом, последовательное выполнение набора модульных тестов требует небольших затрат.

А сквозные тесты ведут себя совершенно иначе. Для их проведения на самом деле требуется написать и выполнить все приложение, что может потребовать немалых затрат вычислительных ресурсов. Перемещение по страницам, установка всего приложения в исходное состояние, инициализация и сброс аутентификации и прочие операции, обычно выполняемые в сквозных тестах, могут отнять немало времени. Таким образом, здесь не имеет особого смысла выполнять сквозные тесты, отслеживая изменения в них с помощью файлового наблюдателя.

См. также

- Рецепт "Создание минимально жизнеспособного набора модульных тестов средствами Karma, Jasmine и TypeScript" с введением в модульные тесты на языке TypeScript.

Модульное тестирование синхронной службы

Типы служб в Angular 2, по существу, являются классами, предназначенными для внедрения. Их нетрудно тестировать, поскольку можно полностью контролировать, как и где они предоставляются, а следовательно, сколько их экземпляров удастся создать. Поэтому тесты служб будут мало чем отличаться от тестов многих обычных классов TypeScript. Прежде чем продолжить дальше, рекомендуется ознакомиться с предыдущими рецептами в этой главе.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/3107/**.

Подготовка

Допустим, требуется создать службу, действующую по принципу "волшебного шара предсказаний". Начните со следующего исходного кода службы, снабженного ради ясности комментариями:

```
[src/app/magic-eight-ball.service.ts]

import {Injectable} from '@angular/core';

@Injectable()
  export class MagicEightBallService {
    private values:Array<string>;
    private lastIndex:number;

    constructor() {
      // инициализировать массив значений,
      // где должно быть не меньше двух элементов
      this.values = [
        'Ask again later',
        'Outlook good',
        'Most likely',
        'Don't count on it'
      ];

      // инициализировать любым достоверным индексом
      this.lastIndex = this.getIndex();
    }

    private getIndex():number {
      // возвратить произвольный индекс по ссылке this.values
      return Math.floor(Math.random() * this.values.length);
    }
    reveal():string {
      // сформировать новый индекс
      let newIdx = this.getIndex();

      // проверить, оказался ли индекс таким же,
      // как и индекс, применявшийся в последний раз
      if (newIdx === this.lastIndex) {
        // Если это так, то произвести сдвиг (перенос)
        // к следующему индексу в массиве. Тем не менее
        // это произвольное поведение
        newIdx = (++newIdx) % this.values.length;
      }
      // сохранить индекс, используемый в настоящий момент
      this.lastIndex = newIdx;

      // получить доступ к символьной строке и возвратить ее
      return this.values[newIdx];
  }
}
```

Поведение данной службы имеет следующие особенности.

- В классе данной службы имеется несколько закрытых членов и только один открытый компонентный метод.
- Данная служба произвольно вызывается из массива.
- Она не должна возвращать одно и то же значение два раза подряд.

Перечисленные выше особенности данной службы следует принимать во внимание при написании модульных тестов, чтобы полностью проверить ее поведение.

Реализация

Начните с создания следующей среды для тестового файла:

```
[src/app/magic-eight-ball.service.spec.ts]

import {TestBed} from '@angular/core/testing';
import {MagicEightBallService}
        from './magic-eight-ball.service';

describe('Service: MagicEightBall', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({
      providers: [
        MagicEightBallService
      ]
    });
  });
});
```

В представленном до сих пор коде нет пока что ничего особенного. Служба типа `MagicEightBallService` вполне внедряема, ее необходимо указать в объявлении модуля, что здесь и сделано. Но для того чтобы воспользоваться данной службой в модульном тесте, придется произвести ее формальное внедрение, поскольку именно это требуется для доступа к ней из компонента. И это можно сделать с помощью функции `inject()`, как выделено ниже полужирным шрифтом.

```
[src/app/magic-eight-ball.service.spec.ts]

import {TestBed, inject} from '@angular/core/testing';
import {MagicEightBallService}
        from './magic-eight-ball.service';

describe('Service: MagicEightBall', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({
      providers: [
```

```
        MagicEightBallService
      ]
    });
  });

  it('should be able to be injected',
     inject([MagicEightBallService],
     (magicEightBallService: MagicEightBallService) => {
       expect(magicEightBallService).toBeTruthy();
     })
  );
});
```

Для начала совсем не плохо, но пока что в данной службе фактически ничего не тестируется. Поэтому напишите выделенный ниже полужирным шрифтом тест, где проверяется, является ли ненулевой возвращаемая символьная строка.

```
[src/app/magic-eight-ball.service.spec.ts]

import {TestBed, inject} from '@angular/core/testing';
import {MagicEightBallService}
        from './magic-eight-ball.service';

describe('Service: MagicEightBall', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({
      providers: [
        MagicEightBallService
      ]
    });
  });

  it('should be able to be injected',
     inject([MagicEightBallService],
     (magicEightBallService: MagicEightBallService) => {
       expect(magicEightBallService).toBeTruthy();
     })
  );

  it('should return a string with nonzero length',
     inject([MagicEightBallService],
     (magicEightBallService: MagicEightBallService) => {
       let result = magicEightBallService.reveal();
       expect(result).toEqual(jasmine.any(String));
       expect(result.length).toBeGreaterThan(0);
     })
  );
});
```

И, наконец, необходимо написать тест, чтобы проверить, не являются ли оба возвращаемых значения одинаковыми. Данная служба вызывается произвольно, и даже если обращаться к ней многократно, то нельзя гарантировать правильность полученного результата. Впрочем, достаточно выполнить проверку 50 раз подряд, чтобы убедиться в правильности работы данной службы, как показано ниже.

[src/app/magic-eight-ball.service.spec.ts]

```
import {TestBed, inject} from '@angular/core/testing';
import {MagicEightBallService}
        from './magic-eight-ball.service';

describe('Service: MagicEightBall', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({
      providers: [
        MagicEightBallService
      ]
    });
  });

  it('should be able to be injected',
     inject([MagicEightBallService],
    (magicEightBallService: MagicEightBallService) => {
      expect(magicEightBallService).toBeTruthy();
    })
  );

  it('should return a string with nonzero length',
     inject([MagicEightBallService],
    (magicEightBallService: MagicEightBallService) => {
      let result = magicEightBallService.reveal();
      expect(result).toEqual(jasmine.any(String));
      expect(result.length).toBeGreaterThan(0);
    })
  );

  it('should not return the same value twice in a row',
     inject([MagicEightBallService],
    (magicEightBallService: MagicEightBallService) => {
      let last;
      for(let i = 0; i < 50; ++i) {
        let next = magicEightBallService.reveal();
        expect(next).not.toEqual(last);
        last = next;
      }
    })
  );
});
```

Замечательно! Все приведенные выше тесты прошли. В итоге вам удалось создать инкрементное и описательное покрытие данной службы тестами.

Принцип действия

Тестовая функция `inject()` автоматически внедряет зависимость всякий раз, когда она вызывается, используя внедряемые классы, передаваемые ей в качестве первого аргумента. А стрелочная функция, передаваемая ей в качестве второго аргумента, поведет себя практически так же, как и конструктор компонента, где в качестве параметра можно указать экземпляр проверяемой службы.

Внедрение зависимостей в данном случае отличается от вызова конструктора компонента, главным образом, тем, что в теле конструктора компонента можно непосредственно сделать ссылку **`this.magicEightBallService`** на данную службу. Что же касается внедрения зависимостей в модульные тесты, то оно не присоединяет автоматически данную службу к ссылке **`this`**.

Дополнение

Для модульного тестирования важно понимать, как должны быть написаны и проведены тесты. При этом существенное значение имеет соблюдение границ открытых и закрытых членов класса. Рассматриваемые здесь тесты написаны таким образом, чтобы в них употреблялись только открытые члены класса, реализующего службу, что дает возможность изменять, расширять или реорганизовывать внутреннюю структуру службы, не особенно беспокоясь о нарушении тестов или необходимости обновить их. Правильно сконструированный класс можно полностью протестировать из его открытого интерфейса.

В связи с приведенным выше замечанием возникает интересный философский вопрос, касающийся модульного тестирования. В частности, поведение вполне сформированной службы должно быть описано как функция ее открытых членов. А если первое утверждение верно, то написать модульные тесты для вполне сформированной службы относительно просто.

Если же написать модульные тесты трудно (например, для этого потребуется надлежащий доступ к закрытому члену проверяемой службы), в таком случае следует принять во внимание, что данная служба, возможно, разработана не так удачно, как следовало было бы. Короче говоря, если служба с трудом поддается тестированию, значит, реализующий ее класс, вероятнее всего, был написан неудачно.

Тестирование без внедрения зависимостей

Наблюдательный читатель заметит, что у тестируемой здесь службы отсутствует сколько-нибудь значимая зависимость после внедрения. Если

внедрить ее в различных местах приложения, то она, безусловно, будет предоставлена согласованно, но в определении службы об этом факте нет ни малейшего представления. Ведь речь в конечном счете идет лишь о получении экземпляра, и данная служба оказывается не более чем внедряемым классом. Следовательно, можно вообще не беспокоиться о внедрении данной службы, получив ее экземпляр с помощью ключевого слова new, как показано ниже. Безусловно, для этого нужно внимательно проследить, насколько важно для службы быть где-нибудь внедренной.

```
[src/app/magic-eight-ball.service.spec.ts]

import {MagicEightBallService}
        from './magic-eight-ball.service';

describe('Service: MagicEightBall', () => {
  let magicEightBallService;
    beforeEach(() => {
      magicEightBallService = new MagicEightBallService();
    });

  it('should be able to be injected', () => {
    expect(magicEightBallService).toBeTruthy();
  });

  it('should return a string with nonzero length', () => {
    let result = magicEightBallService.reveal();
    expect(result).toEqual(jasmine.any(String));
    expect(result.length).toBeGreaterThan(0);
  });

  it('should not return the same value twice in a row', () => {
    let last;
    for(let i = 0; i < 50; ++i) {
      let next = magicEightBallService.reveal();
      expect(next).not.toEqual(last);
      last = next;
    }
  });
});
```

См. также

- Рецепт "Модульное тестирование зависимого от службы компонента с помощью заглушек", в котором демонстрируется, как сымитировать службу для написания модульных тестов, избегая прямых зависимостей.
- Рецепт "Модульное тестирование зависимого от службы компонента с помощью шпионов", где показывается, как отслеживать вызовы служебных методов в модульном тесте.

Модульное тестирование зависимого от службы компонента с помощью заглушек

Протестировать самостоятельный компонент нетрудно, но писать целенаправленные тесты для проверки существующего обособленно компонента приходится крайне редко. Чаще всего у компонента имеется одна или несколько зависимостей, и в этой связи написание удачных модульных тестов отличается от написания неудачных как отрада от отчаяния.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/6651/**.

Подготовка

Допустим, имеется служба, реализованная ранее в рецепте "Модульное тестирование синхронной службы". А кроме того, имеется приведенный ниже компонент, где эта служба применяется. Цель состоит в том, чтобы написать набор модульных тестов для этого компонента, не устанавливая зависимость от данной службы.

```
[src/app/magic-eight-ball/magic-eight-ball.component.ts]

import {Component} from '@angular/core';
import {MagicEightBallService}
        from '../magic-eight-ball.service';

@Component({
  selector: 'app-magic-eight-ball',
  template: `
    <button (click)="update()">Click me!</button>
     <h1>{{ result }}</h1>
  `
})
export class MagicEightBallComponent {
  result: string = '';

  constructor(private magicEightBallService_:
              MagicEightBallService) {}

  update() {
    this.result = this.magicEightBallService_.reveal();
  }
}
```

Реализация

Начните со следующей заготовки тестового файла:

```
[src/app/magic-eight-ball/magic-eight-ball.component.spec.ts]

import {TestBed, async} from '@angular/core/testing';
import {MagicEightBallComponent}
        from './magic-eight-ball.component';
import {MagicEightBallService}
        from '../magic-eight-ball.service';

describe('Component: MagicEightBall', () => {
  beforeEach(async(() => {
  }));

  afterEach(() => {
  });

  it('should begin with no text', async(() => {
  }));

  it('should show text after click', async(() => {
  }));
});
```

Прежде всего необходимо сконфигурировать тестовый модуль, чтобы он правильно предоставлял в тесте импортируемые целевые компоненты, как выделено ниже полужирным шрифтом.

```
[src/app/magic-eight-ball/magic-eight-ball.component.spec.ts]

import {TestBed, async} from '@angular/core/testing';
import {MagicEightBallComponent}
        from './magic-eight-ball.component';
import {MagicEightBallService}
        from '../magic-eight-ball.service';

describe('Component: MagicEightBall', () => {
  let fixture;

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [
        MagicEightBallComponent
      ],
      providers: [
        MagicEightBallService
      ]
    });
```

```
    fixture = TestBed.createComponent(MagicEightBallComponent);
  }));

  afterEach(() => {
    fixture = undefined;
  });

  it('should begin with no text', async(() => {
  }));

  it('should show text after click', async(() => {
  }));
});
```

Имитация зависимости от службы с помощью заглушки

Внедрить конкретную службу, конечно, можно, но ведь в данном случае требуется совсем другое. Вряд ли стоит внедрять экземпляр службы типа `MagicEightBallService` в компонент, если при этом будет установлена зависимость от данной службы, а модульный тест станет более сложным, чем нужно. Но в компонент `MagicEightBallComponent` все равно нужно импортировать *что-то*, похожее на службу типа `MagicEightBallService`. Превосходным вариантом решения данной задачи служит создание заглушки и ее внедрение вместо данной службы в качестве ее имитации, как показано ниже. В итоге компонент не сможет отличить фактическую службу от ее имитации, а следовательно, он будет вести себя как обычно при заданных в тесте условиях.

```
[src/app/magic-eight-ball/magic-eight-ball.component.spec.ts]

import {TestBed, async} from '@angular/core/testing';
import {MagicEightBallComponent}
        from './magic-eight-ball.component';
import {MagicEightBallService}
        from '../magic-eight-ball.service';

describe('Component: MagicEightBall', () => {
  let fixture;
  let magicEightBallResponse = 'Answer unclear';
    let magicEightBallServiceStub = {
    reveal: () => magicEightBallResponse
  };

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [
        MagicEightBallComponent
      ],
```

```
      providers: [
       {
         provide: MagicEightBallService,
         useValue: magicEightBallServiceStub
       }
      ]
    });
    fixture = TestBed.createComponent(MagicEightBallComponent);
  }));

  afterEach(() => {
    fixture = undefined;
  });

  it('should begin with no text', async(() => {
  }));

  it('should show text after click', async(() => {
  }));
});
```

Далее следует написать тест предварительной подготовки, где проверяется, не содержит ли текст свойство `nativeElement` тестовой среды:

```
[src/app/magic-eight-ball/magic-eight-ball.component.spec.ts]

import {TestBed, async} from '@angular/core/testing';
import {MagicEightBallComponent}
        from './magic-eight-ball.component';
import {MagicEightBallService}
        from '../magic-eight-ball.service';

describe('Component: MagicEightBall', () => {
  let fixture;
  let getHeaderEl = () =>
    fixture.nativeElement.querySelector('h1');
  let magicEightBallResponse = 'Answer unclear';
  let magicEightBallServiceStub = {
    reveal: () => magicEightBallResponse
  };

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [
        MagicEightBallComponent
      ],
      providers: [
        {
          provide: MagicEightBallService,
          useValue: magicEightBallServiceStub
```

```
        }
      ]
    });
    fixture = TestBed.createComponent(MagicEightBallComponent);
  }));

  afterEach(() => {
    fixture = undefined;
  });

  it('should begin with no text', async(() => {
    fixture.detectChanges();
    expect(getHeaderEl().textContent).toEqual('');
  }));

  it('should show text after click', async(() => {
  }));
});
```

Инициирование событий в тестовой среде компонента

Во втором тесте следует инициировать событие от щелчка на экранной кнопке, дать тестовой среде команду обнаружить изменение, а затем проверить модель DOM, чтобы выяснить, правильно ли введен текст. А поскольку ранее был определен текст, который возвратит заглушка, то его можно просто сравнить с обнаруженным изменением, как показано ниже. Обратите внимание на то, что для этого требуется импортировать и вызвать предикатный метод `By.css()`, который требуется для проверки объектов типа `DebugElement`.

[src/app/magic-eight-ball/magic-eight-ball.component.spec.ts]

```
import {TestBed, async} from '@angular/core/testing';
import {MagicEightBallComponent}
        from './magic-eight-ball.component';
import {MagicEightBallService}
        from '../magic-eight-ball.service';
import {By} from '@angular/platform-browser';

describe('Component: MagicEightBall', () => {
  let fixture;
  let getHeaderEl = () =>
    fixture.nativeElement.querySelector('h1');
  let magicEightBallResponse = 'Answer unclear';
  let magicEightBallServiceStub = {
    reveal: () => magicEightBallResponse
  };

  ...

  it('should begin with no text', async(() => {
```

```
    expect(getHeaderEl().textContent).toEqual('');
  }));

  it('should show text after click', async(() => {
    fixture.debugElement.query(By.css('button'))
           .triggerEventHandler('click', null);
    fixture.detectChanges();
    expect(getHeaderEl().textContent)
           .toEqual(magicEightBallResponse);
  }));
});
```

Реализация

Как демонстрировалось в главе 7, посвященной внедрению зависимостей, предоставление заглушки компоненту ничем не отличается от предоставления обыкновенного значения базовому приложения. Но здесь заглушка является одиночной функцией, возвращающей статическое значение. В ней отсутствует принцип произвольного выбора из массива символьных строк проверяемой службы, поскольку в этом нет никакой необходимости. Ведь модульные тесты самой службы обеспечивают надлежащее ее поведение. Вместо этого единственным значением, которое здесь предоставляется службой, является информация, передаваемая обратно компоненту для последующей интерполяции в шаблоне.

См. также

- Рецепт "Написание минимально жизнеспособного набора модульных тестов для простого компонента" с элементарным примером модульного тестирования компонентов Angular 2.
- Рецепт "Модульное тестирование синхронной службы", где демонстрируется порядок имитации внедрения зависимостей в модульных тестах.
- Рецепт "Модульное тестирование зависимого от службы компонента с помощью шпионов", в котором показывается, как отслеживать вызовы служебных методов в модульном тесте.

Модульное тестирование зависимого от службы компонента с помощью шпионов

Снабжать службы заглушками, конечно, удобно, но такая возможность ограничивается определенным числом способов. Хотя это может быть не совсем удобно, поскольку создаваемые заглушки должны оставаться актуальными на уровне открытого интерфейса службы. Но в распоряжении

разработчиков имеется еще одно замечательное инструментальное средство, оригинально называемое *шпионом* (spy).

Шпион позволяет выбрать конкретную функцию или метод. Он также помогает собрать информацию, извещающую, что и как вызывалось и как это "нечто" вело себя после вызова. Понятие шпиона подобно заглушке, но оно позволяет создавать намного более надежные модульные тесты.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/3444/**.

Подготовка

Начните со следующих тестов компонента, написанных в предыдущем рецепте:

```
[src/app/magic-eight-ball/magic-eight-ball.component.spec.ts]

import {TestBed, async} from '@angular/core/testing';
import {MagicEightBallComponent}
        from './magic-eight-ball.component';
import {MagicEightBallService}
        from '../magic-eight-ball.service';
import {By} from '@angular/platform-browser';

describe('Component: MagicEightBall', () => {
  let fixture;
  let getHeaderEl = () =>
      fixture.nativeElement.querySelector('h1');
  let magicEightBallResponse = 'Answer unclear';
  let magicEightBallServiceStub = {
     reveal: () => magicEightBallResponse
  };

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [
        MagicEightBallComponent
      ],
      providers: [
        {
          provide: MagicEightBallService,
          useValue: magicEightBallServiceStub
        }
      ]
    });
    fixture = TestBed.createComponent(MagicEightBallComponent);
  }));

  afterEach(() => {
```

```
    fixture = undefined;
  });

  it('should begin with no text', async(() => {
    fixture.detectChanges();
    expect(getHeaderEl().textContent).toEqual('');
  }));

  it('should show text after click', async(() => {
    fixture.debugElement.query(By.css('button'))
           .triggerEventHandler('click', null);
    fixture.detectChanges();
    expect(getHeaderEl().textContent)
           .toEqual(magicEightBallResponse);
  }));
});
```

Принцип действия

Вместо того чтобы воспользоваться заглушкой, сконфигурируйте тестовый модуль, чтобы предоставить конкретную службу, как показано ниже.

[src/app/magic-eight-ball/magic-eight-ball.component.spec.ts]

```
import {TestBed, async} from '@angular/core/testing';
import {MagicEightBallComponent}
        from './magic-eight-ball.component';
import {MagicEightBallService}
        from '../magic-eight-ball.service';
import {By} from '@angular/platform-browser';

describe('Component: MagicEightBall', () => {
  let fixture;
  let getHeaderEl = () =>
      fixture.nativeElement.querySelector('h1');
  let magicEightBallResponse = 'Answer unclear';

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [
        MagicEightBallComponent
      ],
      providers: [
        MagicEightBallService
      ]
    });
    fixture = TestBed.createComponent(MagicEightBallComponent);
  }));

  afterEach(() => {
    fixture = undefined;
  });
```

```
  it('should begin with no text', async(() => {
    fixture.detectChanges();
    expect(getHeaderEl().textContent).toEqual('');
  }));

  it('should show text after click', async(() => {
    fixture.debugElement.query(By.css('button'))
           .triggerEventHandler('click', null);
    fixture.detectChanges();
    expect(getHeaderEl().textContent)
           .toEqual(magicEightBallResponse);
  }));
});
```

Установка шпиона для внедряемой службы

Цель состоит в том, чтобы воспользоваться шпионом и перехватывать вызовы метода `reveal()` для проверяемой службы. Но трудность здесь состоит в том, что проверяемая служба внедряется в компонент. Следовательно, у вас отсутствует возможность передать получение ссылки экземпляру службы и установить для нее шпион. Правда, тестовая среда компонента сделает это за вас следующим образом:

```
[src/app/magic-eight-ball/magic-eight-ball.component.spec.ts]

import {TestBed, async} from '@angular/core/testing';
import {MagicEightBallComponent}
        from './magic-eight-ball.component';
import {MagicEightBallService}
        from '../magic-eight-ball.service';
import {By} from '@angular/platform-browser';

describe('Component: MagicEightBall', () => {
  let fixture;
  let getHeaderEl = () =>
      fixture.nativeElement.querySelector('h1');
  let magicEightBallResponse = 'Answer unclear';
  let magicEightBallService;

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [
        MagicEightBallComponent
      ],
      providers: [
        MagicEightBallService
      ]
    });
    fixture = TestBed.createComponent(MagicEightBallComponent);
    magicEightBallService = fixture.debugElement.injector
```

```
        .get(MagicEightBallService);
  }));

  afterEach(() => {
    fixture = undefined;
    magicEightBallService = undefined;
  });
  ...
});
```

Далее установите шпион для экземпляра службы, используя метод spyOn(). Сконфигурируйте шпион, чтобы перехватывать вызов метода и возвращать вместо этого статическое значение, как выделено ниже полужирным шрифтом.

```
[src/app/magic-eight-ball/magic-eight-ball.component.spec.ts]

import {TestBed, async} from '@angular/core/testing';
import {MagicEightBallComponent}
        from './magic-eight-ball.component';
import {MagicEightBallService}
        from '../magic-eight-ball.service';
import {By} from '@angular/platform-browser';

describe('Component: MagicEightBall', () => {
  let fixture;
  let getHeaderEl = () =>
      fixture.nativeElement.querySelector('h1');
  let magicEightBallResponse = 'Answer unclear';
  let magicEightBallService;
  let revealSpy;

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [
        MagicEightBallComponent
      ],
      providers: [
        MagicEightBallService
      ]
    });
    fixture = TestBed.createComponent(MagicEightBallComponent);
    magicEightBallService = fixture.debugElement.injector
        .get(MagicEightBallService);
    revealSpy = spyOn(magicEightBallService, 'reveal')
        .and.returnValue(magicEightBallResponse);
  }));

  afterEach(() => {
    fixture = undefined;
```

```
    magicEightBallService = undefined;
    revealSpy = undefined;
  });
  ...
});
```

С помощью такого шпиона можно теперь выяснить, каким образом остальная часть приложения взаимодействует с перехваченным методом. Добавьте далее новый тест, где проверяется, был ли данный метод вызван лишь один раз и возвратил ли он правильное значение после щелчка кнопкой мыши. Это способствует также переносу действия, совершаемого при щелчке кнопкой мыши, в собственный тестовый вспомогательный метод:

[src/app/magic-eight-ball/magic-eight-ball.component.spec.ts]

```
import {TestBed, async} from '@angular/core/testing';
import {MagicEightBallComponent}
        from './magic-eight-ball.component';
import {MagicEightBallService}
        from '../magic-eight-ball.service';
import {By} from '@angular/platform-browser';

describe('Component: MagicEightBall', () => {
  let fixture;
  let getHeaderEl = () =>
      fixture.nativeElement.querySelector('h1');
  let magicEightBallResponse = 'Answer unclear';
  let magicEightBallService;
  let revealSpy;

  let clickButton = () => {
    fixture.debugElement.query(By.css('button'))
           .triggerEventHandler('click', null);
  };

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [
        MagicEightBallComponent
      ],
      providers: [
        MagicEightBallService
      ]
    });
    fixture = TestBed.createComponent(MagicEightBallComponent);
    magicEightBallService = fixture.debugElement.injector
         .get(MagicEightBallService);
    revealSpy = spyOn(magicEightBallService, 'reveal')
         .and.returnValue(magicEightBallResponse);
  }));
```

```
  afterEach(() => {
    fixture = undefined;
    magicEightBallService = undefined;
    revealSpy = undefined;
  });

  it('should begin with no text', async(() => {
    fixture.detectChanges();
    expect(getHeaderEl().textContent).toEqual('');
  }));

  it('should call reveal after a click', async(() => {
  clickButton();
  expect(revealSpy.calls.count()).toBe(1);
  expect(revealSpy.calls.mostRecent().returnValue)
        .toBe(magicEightBallResponse);
  }));

  it('should show text after click', async(() => {
    clickButton();
    fixture.detectChanges();
    expect(getHeaderEl().textContent)
          .toEqual(magicEightBallResponse);
  }));
});
```

Обратите внимание на то, что метод **detectChanges()** требуется лишь для разрешения привязки данных, а не для выполнения обработчиков событий.

Принцип действия

Шпионы действуют в Jasmine как перехватчики методов, способные проверять все, что связано с вызовом данного метода. В частности, шпион может отслеживать, как и когда был вызван метод, какие аргументы были переданы при его вызове, сколько раз он вызывался, каким образом он должен был себя вести и т.д. И это оказывается очень удобно при попытке удалить зависимости из модульных тестов компонента, поскольку дает возможность сымитировать открытый интерфейс службы, используя шпионы.

Дополнение

Шпионы не призваны заменить собой полностью методы. Здесь шпион оказывается удобным для того, чтобы выполнение не достигло внутренней структуры службы. Но не трудно себе представить случаи, когда требуется лишь пассивно наблюдать за вызовом определенного метода, не нарушая нормальное продолжение выполнения. Для подобных целей вместо вызова метода `.and.returnValue()` можно воспользоваться вызовом доступного в

Jasmine метода `.and.callThrough()`, чтобы обеспечить непрерывность выполнения.

См. также

- Рецепт "Написание минимально жизнеспособного набора модульных тестов для простого компонента" с элементарным примером модульного тестирования компонентов Angular 2.
- Рецепт "Модульное тестирование синхронной службы", где демонстрируется порядок имитации внедрения зависимостей в модульных тестах.
- Рецепт "Модульное тестирование зависимого от службы компонента с помощью заглушек", в котором показывается, как сымитировать службу для написания модульных тестов, избегая прямых зависимостей.

Глава 10

Производительность и развитые концепции

В этой главе даются следующие рецепты.

- Общее представление о функции `enableProdMode()` и надлежащем ее применении в чистых и нечистых каналах
- Работа с зонами за пределами Angular
- Прием событий от службы типа `NgZone`
- Выполнение за пределами зоны Angular
- Настройка компонентов на явное обнаружение изменений с помощью стратегии `OnPush`
- Настройка инкапсуляции представлений на максимальную производительность
- Настройка службы типа `Renderer` на применение рабочих веб-процессов
- Настройка приложений на компиляцию перед выполнением
- Настройка приложений на загрузку по требованию

Введение

Каркас Angular 2 был полностью переделан по многим причинам, но одной из главных причин, безусловно, была производительность. Этот восставший из пепла каркас выглядит весьма утонченным и изящным, хотя он не лишен сложностей. В этой главе представлены новые средства, положенные в основу Angular 2, а также поясняется, как пользоваться ими наиболее эффективно для рационализации приложений.

Общее представление о функции enableProdMode() и надлежащем ее применении в чистых и нечистых каналах

Применяемый в Angular 2 процесс обнаружения изменений изящен, но может быть превратно понят с первого раза. Несмотря на то что данный процесс обеспечивает заметный выигрыш в производительности по сравнению с версией Angular 1.x, такой выигрыш достигается совсем не бесплатно. По умолчанию в Angular активизируется режим разработки, где вас предупреждают, что поведение вашего кода таит в себе опасность свести на нет все преимущества, достигаемые в эффективности обнаружения изменений. В данном рецепте вам предстоит реализовать средство, нарушающее принятую в Angular схему обнаружения изменений, внести в него соответствующие коррективы и благополучно воспользоваться функцией `enableProdMode()`.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/0623/**.

Подготовка

Начните со следующего простого компонента:

```
[src/app/app.component.ts]

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

@Component({
  selector: 'app-root',
  template: `
    <input #t>
    <button (click)="update(t.value)">Save</button>
    <h1>{{ title }}</h1>
  `
})
export class AppComponent {
  title:string = '';

  update(newTitle:string) {
    this.title = newTitle;
  }
}
```

При нажатии экранной кнопки в данном компоненте, значение из поля ввода данных извлекается и вставляется в дескриптор разметки заголовка. В таком виде реализация данного компонента оказывается вполне применимой.

Реализация

Чтобы продемонстрировать целесообразность применения функции `enableProdMode()`, необходимо пояснить поведение, которое она маскирует. Это, в частности, означает, что некоторая часть приложения будет вести себя по-разному, когда обнаруживаются два изменения подряд.

Реализовать такое поведение можно самыми разными способами, но в данном рецепте будет реализован бессмысленный канал, изменяющийся всякий раз, когда он используется.

Возникновение ошибки согласованности

Создайте бессмысленный канал типа `AddRandomPipe` следующим образом:

```
[src/app/add-random.pipe.ts]

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

@Pipe({
  name: 'addRandomPipe'
})
export class AddRandomPipe implements PipeTransform {
  transform(value:string):string {
    return value + Math.random();
  }
}
```

Затем внедрите этот канал в свой компонент, как выделено ниже полужирным шрифтом. Но это пока еще не приведет к ошибке.

```
[src/app/app.component.ts]

import {Component} from '@angular/core';
import {AddRandomPipe} from './add-random.pipe';

@Component({
  selector: 'app-root',
  template: `
    <input #t>
    <button (click)="update(t.value)">Save</button>
    <h1>{{ title | addRandomPipe }}</h1>
  `
})
export class AppComponent {
  title:string = '';

  update(newTitle:string) {
    this.title = newTitle;
  }
}
```

Процесс обнаружения изменений действительно будет выполнен в Angular дважды, и поэтому может показаться странным, почему это не приводит к ошибке. Несмотря на то что канал будет формировать новые выходные данные всякий раз, когда вызывается его метод **transform()**, Angular обладает достаточно развитой логикой, чтобы распознать, что входные данные, предназначенные для вставки в дескриптор разметки, не изменились, а следовательно, повторное вычисление канала не требуется. Такой канал называется "чистым" и устанавливается в Angular по умолчанию.

Чтобы вынудить Angular вычислять канал в каждом цикле обнаружения изменений, укажите его как нечистый следующим образом:

```
[src/app/add-random.pipe.ts]

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

@Pipe({
  name: 'addRandomPipe',
  pure: false
})
export class AddRandomPipe implements PipeTransform {
  transform(value:string):string {
    return value + Math.random();
  }
}
```

А теперь начинается самое интересное. Запустив свое приложение на выполнение, вы должны обнаружить сообщение об ошибке, аналогичное приведенному ниже.

```
EXCEPTION: Error in ./AppComponent class AppComponent - inline
template:3:8
caused by: Expression has changed after it was checked. Previous value:
'0.0495151713435904'. Current value: '0.9266277919907477'.
(ИСКЛЮЧЕНИЕ: Ошибка в шаблоне :3:8, встроенном в класс ./AppComponent,
 вызвана тем, что выражение изменилось после его проверки.
 Предыдущее значение: '0.0495151713435904'.
 Текущее значение:    '0.9266277919907477'.)
```

Обеспечение соответствия обнаруженным изменениям

Приведенная выше ошибка возникнет, как только приложение запустится на выполнение, и это произойдет прежде чем, у вас вообще появится возможность нажать экранную кнопку. Это означает, что Angular обнаружит несоответствие привязки данных, как только будет получен и визуализирован в первый раз экземпляр компонента.

Итак, вы создали канал, который намеренно изменяется всякий раз. Следовательно, цель состоит в том, чтобы дать Angular команду не проверять данные, выводимые из канала, на соответствие дважды после получения экземпляра компонента.

На данном этапе вам удалось вынудить Angular выдать ошибку согласованности, которая возникает в связи с тем, что при выполнении дважды цикла обнаружения изменений получаются разные результаты проверки. Если вызвать в этот момент функцию `enableProdMode()`, ошибка не будет выдана, поскольку цикл обнаружения изменений будет выполнен в Angular лишь один раз без проверки на соответствие. И это происходит потому, что Angular доверяет тому, что вы выполнили проверку на соответствие, прежде чем вызвать функцию `enableProdMode()`. Установка рабочего режима с помощью функции `enableProdMode()` для маскирования сообщений об ошибке согласованности сродни ситуации, когда вы, придя домой, обнаруживаете в нем пожар и решаете отправиться в отпуск.

Контролировать подобную ситуацию в Angular можно, задавая соответствующую стратегию обнаружения изменений для компонента. По умолчанию проверка при обнаружении изменений производится всегда, но это положение можно изменить, задав в конфигурации компонента стратегию `OnPush`, как выделено ниже полужирным шрифтом. Теперь ошибка согласованности не должна возникнуть при инициализации экземпляра компонента.

```
[src/app/app.component.ts]

import {Component, ChangeDetectionStrategy }
       from '@angular/core';
import {AddRandomPipe} from './add-random.pipe';

@Component({
  selector: 'app-root',
  template: `
    <input #t>
    <button (click)="update(t.value)">Save</button>
    <h1>{{ title | addRandomPipe }}</h1>
  `,
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class AppComponent {
  title:string = '';

  update(newTitle:string) {
    this.title = newTitle;
  }
}
```

Переход в рабочий режим

Теперь ваше приложение совместимо с действующим в Angular механизмом обнаружения изменений, и поэтому вы вольны воспользоваться функцией `enableProdMode()`. Это дает вашему приложению возможность всякий раз выполнять цикл обнаружения изменений лишь один раз, поскольку предполагается, что приложение придет в устойчивое состояние.

Вызовите функцию `enableProdMode()` в файле начальной загрузки приложения, прежде чем запустить начальную загрузку:

```
[src/main.ts]

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

enableProdMode();

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

Принцип действия

Функция `enableProdMode()` сконфигурирует ваше приложение разными способами, включая подавление ошибок и информационных сообщений об ошибках. Очевидно, что ни один из этих способов не подавляет вторичный процесс обнаружения изменений.

Дополнение

Имеются и другие способы устранить проблему согласованности. Допустим, требуется сформировать канал, в котором будет добавляться случайное число к выходным данным. И каждый раз это совсем не обязательно должно быть другое случайное число. Такое число должно быть назначено каждому уникальному набору входных данных. В подобной ситуации можно создать канал, где применяется определенного рода стратегия кеширования. Так, если в канале кешируются данные, полученные за период времени, превосходящий время, необходимое для завершения цикла обнаружения изменений, который длится не очень долго, то изменять стратегию обнаружения изменений не нужно, поскольку несколько обращений к каналу дадут одинаковый ответ. Рассмотрим в качестве примера следующий фрагмент кода:

```
[src/app/add-random.pipe.ts]

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

```
@Pipe({
  name: 'addRandomPipe',
  pure: false
})

export class AddRandomPipe implements PipeTransform {
  cache:Object = {};

  transform(input:string):string {
    let value = this.cache[input];
    if (!value || value.expire < Date.now()) {
      value = {
        text: input + Math.random(),
        // Истекает через одну секунду
        expire: Date.now() + 1000
      }
      this.cache[input] = value;
    }
    return value.text;
  }
}
```

Таким образом, вы можете благополучно удалить определение стратегии обнаружения изменений `changeDetection: ChangeDetectionStrategy.OnPush` из метаданных в объекте типа `ComponentMetadata` своего компонента типа `AppComponent` и больше не увидеть сообщения об ошибках согласованности.

См. также

- Рецепт "Настройка службы типа `Renderer` на применение рабочих веб-процессов", где поясняется весь процесс настройки приложения для визуализации в рабочем веб-процессе.
- Рецепт "Настройка приложений на компиляцию перед выполнением", в котором поясняется, как скомпилировать приложение во время сборки.

Работа с зонами за пределами Angular

При работе с зонами полностью в пределах каркаса Angular скрывается то, что на самом деле происходит внутри него. Но было бы неуважением к читателю обойти молчанием этот внутренний механизм. Поэтому в данном рецепте обыкновенная реализация библиотеки `zone.js` будет вынесена за пределы Angular и немного модифицирована, чтобы выяснить, как воспользоваться ей в Angular.

В данном рецепте средства Angular вообще не применяются, а вместо них применяются средства библиотеки `zone.js`, включенной в HTML-файл. Более того, исходный код для данного рецепта будет написан ради простоты на языке JavaScript в стандарте ES5.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/0591/**.

Подготовка

Начните с простого HTML-файла со следующим содержимым:

```
[index.html]

<button id="button">Click me</button>
<button id="add">Add listener</button>
<button id="remove">Remove listener</button>

<script>
var button = document.getElementById('button');
var add = document.getElementById('add');
var remove = document.getElementById('remove');

var clickCallback = function() {
  console.log('click!');
};

setInterval(function() {
  console.log('set interval!');
}, 1000);
add.addEventListener('click', function() {
  button.addEventListener('click', clickCallback);
});
remove.addEventListener('click', function() {
  button.removeEventListener('click', clickCallback);
});
</script>
```

В каждом из приведенных выше обратных вызовов имеются операторы протоколирования с функцией `log()`, чтобы было видно, где эти вызовы происходят, как описано ниже.

- Метод `setInterval()` вызывает связанный с ним приемник событий каждую секунду.
- В результате щелчка на экранной кнопке Click me (Щелкните на мне) вызывается связанный с ней приемник событий, если он присоединен к ней.

- В результате щелчка на экранной кнопке Add listener (Добавить приемник событий) к данной кнопке присоединяется приемник событий от щелчков на ней.
- В результате щелчка на экранной кнопке Remove listener (Удалить приемник событий) удаляется приемник событий от щелчков на данной кнопке.

Ничего особенного здесь не происходит потому, что все эти вызовы являются стандартными методами, входящими в API браузера. Но, как станет ясно в дальнейшем, прелесть зон в том и состоит, что поведение зоны можно внедрить вокруг всего этого, вообще не модифицируя код.

Реализация

Прежде всего добавьте файл со сценариями `zone.js` в самом начале созданного ранее HTML-файла, как показано ниже.

```
[index.html]

<script
src="https://cdnjs.cloudflare.com/ajax/libs/
          ↳zone.js/0.6.26/zone.js">
</script>

<button id="button">Click me</button>
<button id="add">Add listener</button>
<button id="remove">Remove listener</button>
...
```

Для файла **`zone.js`** не требуется никаких специальных настроек, но его необходимо включить в HTML-документ перед кодом установки приемников событий или же выполнения любых асинхронных операций. Поскольку каркас Angular зависит от файла **`zone.js`**, то последний необходимо включить перед кодом инициализации Angular.

Благодаря добавлению данного сценария в глобальное пространство внедряется класс `Zone`. А глобальная зона уже создана автоматически и доступна по следующей ссылке:

```
Zone.current
```

Разветвление зоны

В глобальной зоне пока еще не делается ничего интересного. Чтобы специально настроить зону, придется создать собственную зону как разветвление уже имеющейся и выполнить в ней соответствующий код. С этой целью внесите в упомянутый выше HTML-файл изменения, выделенные ниже

полужирным шрифтом. По своему поведению эти изменения ничего не меняют с точки зрения консоли. В частности, методу `fork()` передается пустой объектный литерал типа `ZoneSpec`, который предстоит модифицировать в дальнейшем.

```
[index.html]

<script
  src="https://cdnjs.cloudflare.com/ajax/libs/
            ↳zone.js/0.6.26/zone.js">
</script>

<button id="button">Click me</button>
<button id="add">Add listener</button>
<button id="remove">Remove listener</button>

<script>
var button = document.getElementById('button');
var add = document.getElementById('add');
var remove = document.getElementById('remove');

Zone.current.fork({}).run(function() {
  var clickCallback = function() {
    console.log('click!');
  };

  setInterval(function() {
    console.log('set interval!');
  }, 1000);
  add.addEventListener('click', function() {
    button.addEventListener('click', clickCallback);
  });
  remove.addEventListener('click', function() {
    button.removeEventListener('click', clickCallback);
  });
});
</script>
```

Переопределение событий в зоне средствами класса ZoneSpec

Когда фрагмент кода выполняется в зоне, появляется возможность присоединять нужные виды поведения в важных точках потока асинхронного поведения. В данном случае переопределяются следующие четыре события в зоне:

- `scheduleTask`
- `invokeTask`

- hasTask
- cancelTask

Начните с события scheduleTask. Объявите метод, переопределяемый в классе ZoneSpec, как показано ниже. При определении методов употребляются имена событий, снабженные префиксом on.

```
[index.html]

<script
  src="https://cdnjs.cloudflare.com/ajax/libs/
            ↳zone.js/0.6.26/zone.js">
</script>

<button id="button">Click me</button>
<button id="add">Add listener</button>
<button id="remove">Remove listener</button>

<script>
var button = document.getElementById('button');
var add = document.getElementById('add');
var remove = document.getElementById('remove');

Zone.current.fork({
  onScheduleTask: function(zoneDelegate, zone,
                           targetZone, task) {
    console.log('schedule');
    zoneDelegate.scheduleTask(targetZone, task);
  }
}).run(function() {
  var clickCallback = function() {
    console.log('click!');
  };

  setInterval(function() {
    console.log('set interval!');
  }, 1000);
  add.addEventListener('click', function() {
    button.addEventListener('click', clickCallback);
  });
  remove.addEventListener('click', function() {
    button.removeEventListener('click', clickCallback);
  });
});
</script>
```

Благодаря этому дополнению должно стать ясно, что в зоне распознаются три планируемые задачи. И в этом должен быть свой смысл, поскольку в данном случае объявляются три экземпляра, способные сфор-

мировать асинхронные действия setInterval, addEventListener и removeEventListener. А если щелкнуть на кнопке Add listener, то будет запланирована и четвертая задача.

Вызов метода **zoneDelegate.scheduleTask()** требуется потому, что в данном случае фактически перезаписывается информация, используемая в зоне для обработки планируемых задач. Если не сделать этого, обработчик планируемых задач завершится прежде, чем задача будет запланирована.

Отмена задачи совершенно противоположна ее планированию. Поэтому переопределите далее обработчик событий следующим образом:

```
[index.html]

Zone.current.fork({
  onScheduleTask: function(zoneDelegate, zone,
                           targetZone, task) {
    console.log('schedule');
    zoneDelegate.scheduleTask(targetZone, task);
  },
  onCancelTask: function(zoneDelegate, zone,
                         targetZone, task) {
    console.log('cancel');
    zoneDelegate.cancelTask(targetZone, task);
  }
}).run(function() {
  ...
});
```

Событие отмены наступает, когда удаляется запланированная задача. В данном случае это происходит при вызове метода removeEventListener(). Если щелкнуть сначала на кнопке Add listener, а затем на кнопке Remove listener, то наступит событие отмены.

Планирование задач становится видимым на стадии запуска, но всякий раз, когда нажимается экранная кнопка или выполняется обработчик действий типа setInterval, вообще ничего не протоколируется. Дело в том, что планирование задачи, происходящее во время регистрации, отличается от вызова задачи, который происходит в тот момент, когда фактически наступает асинхронное событие.

Чтобы убедиться в этом, добавьте переопределяемый метод invokeTask(), как выделено ниже полужирным шрифтом. Благодаря этому дополнению должно стать ясно, что вывод протокольного сообщения на консоль происходит всякий раз, когда вызывается задача: в связи с нажатием экранной кнопки или обратным вызовом обработчика действий типа setInterval.

```
[index.html]

Zone.current.fork({
  onScheduleTask: function(zoneDelegate, zone,
                           targetZone, task) {
    console.log('schedule');
    zoneDelegate.scheduleTask(targetZone, task);
  },
  onCancelTask: function(zoneDelegate, zone,
                         targetZone, task) {
    console.log('cancel');
    zoneDelegate.cancelTask(targetZone, task);
  },
  onInvokeTask: function(zoneDelegate, zone, targetZone, task,
                         applyThis, applyArgs) {
    console.log('invoke');
    zoneDelegate.invokeTask(targetZone, task,
                            applyThis, applyArgs);
  }
}).run(function() {
  ...
});
```

До сих пор можно было увидеть, когда именно задачи планируются и вызываются в зоне. Но теперь требуется совершенно противоположное действие, чтобы обнаружить момент, когда все задачи окажутся завершенными. И этого можно добиться с помощью метода `hasTask()`, который также допускает переопределение:

```
[index.html]

Zone.current.fork({
  onScheduleTask: function(zoneDelegate, zone,
                           targetZone, task) {
    console.log('schedule');
    zoneDelegate.scheduleTask(targetZone, task);
  },
  onCancelTask: function(zoneDelegate, zone,
                         targetZone, task) {
    console.log('cancel');
    zoneDelegate.cancelTask(targetZone, task);
  },
  onInvokeTask: function(zoneDelegate, zone, targetZone,
                         task, applyThis, applyArgs) {
    console.log('invoke');
    zoneDelegate.invokeTask(targetZone, task,
                            applyThis, applyArgs);
  },
```

```
  onHasTask: function(zoneDelegate, zone,
                      targetZone, isEmpty) {
    console.log('has');
    zoneDelegate.hasTask(targetZone, isEmpty);
  }
}).run(function() {
  ...
});
```

У объекта `isEmpty`, передаваемого методу `onHasTask()` в качестве параметра, имеются три свойства, `eventTask`, `macroTask` и `microTask`, которые приводятся к логическим значениям, обозначающим, имеются ли задачи в соответствующих очередях.

С помощью приведенных выше четырех обратных вызовов можно успешно перехватить следующие важные моменты в жизненном цикле компонента.

- Когда планируется "генератор" задач, способный формировать задачи, исходя из событий, наступающих в браузере или таймере.
- Когда отменяется "генератор" задач.
- Когда фактически вызывается задача.
- Когда определяется, запланирована ли какая-нибудь задача и какого типа.

Принцип действия

Ядро зон формируется по принципу перехвата асинхронных задач. В частности, требуется выяснить, когда создаются асинхронные задачи, каково их количество и когда они выполняются.

В библиотеке `zone.js` это достигается вставкой в виде прокладки всех соответствующих браузерных методов, отвечающих за установку асинхронных задач. В действительности все методы, применяемые в данном случае, включая `setInterval()`, `addEventListener()` и `removeEventListener()`, вставляются в виде прокладки, чтобы в той зоне, где они выполняются, было известно о любых асинхронных задачах, которые они могут поместить в очередь.

Дополнение

Чтобы каким-то образом связать зоны с Angular, необходимо сделать небольшое отступление и исследовать экосистему зон.

Общее преставление о методе `zone.run()`

Обратите внимание на то, что в данном рецепте результат вызова выводится на консоль для каждого асинхронного действия — даже для тех из них, которые зарегистрированы в другом асинхронном действии. В этом, собственно,

и кроется истинный потенциал зон. Все, что делается в теле метода `zone.run()`, происходит каскадом в пределах одной и той же зоны. Подобным образом в зоне можно отследить любой ненарушенный сегмент асинхронного поведения, не прибегая к написанию целой массы стереотипного кода.

Микро- и макрозадачи

Данный вопрос имеет отношение не к библиотеке `zone.js`, а скорее к тому, как действует цикл ожидания событий в браузере. Все события, формируемые в данном рецепте, в том числе от щелчков кнопками мыши, в таймере и прочие, относятся к макрозадачам. Это означает, что обработчики подобных событий с точки зрения браузера являются синхронными, что приводит к блокированию соответствующего сегмента кода. А код, выполняющийся вокруг этих задач, в том числе обратные вызовы из библиотеки `zone.js`, относится к микрозадачам. Эти задачи отличаются от макрозадач, но они по-прежнему выполняются синхронно как часть всего цикла данной макрозадачи.

Макрозадача может сформировать в своем цикле дополнительные задачи для самой себя.

Как только очереди микро- и макрозадач опустеют, зона может считаться устойчивой, поскольку никакого асинхронного поведения больше не предвидится. Для Angular это самый удобный момент, чтобы обновить пользовательский интерфейс.

В действительности именно это и делается во внутреннем механизме Angular. Каркас Angular смотрит на браузер сквозь "очки" библиотеки `zone.js`, сфокусированные на решении задач, используя это искусное инструментальное средство для того, чтобы решить, когда именно следует перейти к визуализации.

См. также

- Рецепт "Прием событий от службы типа `NgZone`", где дается общее представление о том, как зоны применяются в Angular.
- Рецепт "Выполнение за пределами зоны Angular", в котором показывается, как выполнять долгосрочные операции, не требуя дополнительных издержек на зоны.
- Рецепт "Настройка компонентов на явное обнаружение изменений с помощью стратегии `OnPush`", где поясняется, как управлять вручную процессом обнаружения изменений в Angular.

Прием событий от службы типа NgZone

С появлением Angular 2 возникло и понятие зон. Прежде чем приступить к данному рецепту, настоятельно рекомендуется проработать предыдущий рецепт "Работа с зонами за пределами Angular".

Библиотека zone.js

Каркас Angular 2 непосредственно зависит от библиотеки `zone.js`. Она дает Angular возможность опираться на зону, а следовательно, этот каркас может непосредственно управлять контекстом своего выполнения.

Проще говоря, это означает, что каркас Angular способен различать происходящие асинхронные действия, которые могут его заинтересовать. Если это покажется вам несколько похожим на конструкцию `$scope.apply()` в версии Angular 1.x, то ваше предположение верно.

Служба типа NgZone

Интеграция Angular 2 с зонами принимает форму службы типа `NgZone`, которая действует подобно оболочке в отношении зон в Angular. Эта служба обеспечивает доступ к полезному интерфейсу API, которым можно выгодно воспользоваться.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/8676/**.

Подготовка

Все, что требуется для данного рецепта, — это следующий компонент, в который может быть внедрена служба типа `NgZone`:

```
[src/app/app.component.ts]

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

@Component({
  selector: 'app-root',
  template: ``
})
export class AppComponent {
  constructor() {
  }
}
```

Реализация

Начните с внедрения службы типа NgZone в свой компонент, который становится доступным в базовом модуле Angular, как выделено ниже полужирным шрифтом.

```
[src/app/app.component.ts]

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

@Component({
  selector: 'app-root',
  template: ``
})
export class AppComponent {
  constructor(private zone:NgZone) {
  }
}
```

Служба типа NgZone предоставляет доступ к целому ряду отправителей событий типа EventEmitter, которые можно присоединять. Зоны способны отслеживать происходящие в них асинхронные действия, и поэтому служба типа NgZone предоставляет доступ к двум отправителям событий типа EventEmitter, распознающим моменты, когда асинхронные действия ставятся в очередь и извлекаются из нее как микрозадачи.

В частности, отправитель событий типа EventEmitter в свойстве onUnstable позволяет выяснить, когда в очередь поставлена одна или больше микрозадач, а отправитель событий в свойстве onStable активизируется, когда очередь микрозадач оказывается пустой и Angular не собирается больше ничего в нее ставить.

Добавьте обработчики этих событий, как выделено ниже полужирным шрифтом.

```
[src/app/app.component.ts]

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

@Component({
  selector: 'app-root',
  template: ``
})
export class AppComponent {
  constructor(private zone:NgZone) {
    zone.onStable.subscribe(() => console.log('stable'));
    zone.onUnstable.subscribe(() => console.log('unstable'));
  }
}
```

Замечательно! Хотя вывод протокольных сообщений выглядит скучновато. При запуске приложения на выполнение сообщается, что лишь оно работает устойчиво, но и только.

Демонстрация жизненного цикла зоны

Если вам понятно, каким образом в Angular применяются зоны, то вас не должно удивить недостаточно подробное протоколирование, поскольку в данной зоне нечему формировать асинхронные задачи. Поэтому создайте экранную кнопку с обработчиком событий, где задается оператор протоколирования по истечении времени ожидания, как показано ниже.

```
[src/app/app.component.ts]

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

@Component({
  selector: 'app-root',
  template: `<button (click)="foo()">foo</button>`
})
export class AppComponent {
  constructor(private zone:NgZone) {
    zone.onStable.subscribe(() => console.log('stable'));
    zone.onUnstable.subscribe(() => console.log('unstable'));
  }
  foo() {
    setTimeout(() => console.log('timeout handler'), 1000);
  }
}
```

Теперь после каждого щелчка на экранной кнопке должна появиться пара сообщений о том, что приложение работает сначала неустойчиво, затем — устойчиво, а по истечении одной секунды — неустойчиво, далее сработал обработчик событий, и снова оно работает устойчиво. Это означает, что вы успешно осуществили привязку к отправителям событий в зоне Angular.

Принцип действия

Чтобы не прибегать к конструкции `$scope.apply()`, в Angular должна быть возможность принять логичное решение, когда следует проверить, изменилось ли состояние приложения.

В такой асинхронной среде, как браузер, подобное решение может поначалу показаться неблагодарным делом. Ведь сам характер событий с периодическим отчетом времени и асинхронным вводом данных затрудняет их отслеживание. Рассмотрим в качестве примера следующий фрагмент кода:

```
element.addEventListener('click', _ => {
  // сделать что-нибудь
  setInterval(_ => {
    // сделать что-нибудь
  }, 1000);
});
```

В этом фрагменте кода можно изменить модель в двух местах, причем в обоих асинхронно. Подобный код встречается постоянно и во стольких местах, что трудно себе представить, каким образом отслеживать его, не распространив повсюду конструкцию вроде `$scope.$apply()`.

Применение библиотеки `zone.js`

Зоны примечательны тем, что они дают вам возможность уяснить, как и когда браузер выполняет интересующие вас асинхронные действия. Служба типа `NgZone` автоматически заключает в оболочку базовый интерфейс API вместо того, чтобы предоставлять отправители событий типа `EventEmitter` на разных стадия жизненного цикла зоны, но это не должно вас смущать. В данном рецепте выводимые на консоль протокольные сообщения демонстрируют следующее.

1. Приложение инициализируется и проверяет свою зону. В данный момент запланированные задачи отсутствуют, поэтому Angular отправляет событие, наступающее в связи с устойчивой работой приложения.
2. Вы нажимаете экранную кнопку.
3. Инициируется событие от щелчка на экранной кнопке, которое, в свою очередь, инициирует в зоне задачу для выполнения обработчика событий от щелчков на экранной кнопке. Angular обнаруживает это и отправляет событие, наступающее в связи с неустойчивой работой приложения.
4. Выполняется обработчик событий от щелчков на экранной кнопке, планируя задачу через одну секунду.
5. Обработчик событий от щелчков на экранной кнопке завершает свое выполнение, и в приложении снова отсутствуют ожидающие задачи. Поэтому Angular отправляет событие, наступающее в связи с устойчивой работой приложения.
6. По истечении одной секунды таймер браузера добавляет обработчик событий `setTimeout()` в очередь задач. А поскольку он вставляется как прокладка в зоне, то Angular обнаруживает это и отправляет событие, наступающее в связи с неустойчивой работой приложения.
7. Выполняется обработчик событий `setTimeout()`.

8. Обработчик событий `setTimeout()` завершает свое выполнение, и в приложении снова отсутствуют ожидающие задачи. Поэтому Angular отправляет событие, наступающее в связи с устойчивой работой приложения.

См. также

- Рецепт "Работа с зонами за пределами Angular" с пояснениями, каким образом зоны действуют в браузере.
- Рецепт "Выполнение за пределами зоны Angular", в котором показывается, как выполнять долгосрочные операции, не требуя дополнительных издержек на зоны.
- Рецепт "Настройка компонентов на явное обнаружение изменений с помощью стратегии `OnPush`", где поясняется, как управлять вручную процессом обнаружения изменений в Angular.

Выполнение за пределами зоны Angular

Возможности библиотеки `zone.js` превосходны, поскольку она делает все автоматически, но опытные разработчики программного обеспечения хорошо знают, что за любые удобства нередко приходится платить известную цену. И это особенно справедливо, когда в силу вступает принцип привязки данных. Поэтому в данном рецепте поясняется, как организовать выполнение за пределами зоны Angular и какие выгоды это сулит.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **`http://ngcookbook.herokuapp.com/3362/`**.

Реализация

Чтобы сравнить выполнение в разных контекстах, создайте две экранные кнопки, выполняющие один и тот же код в контекстах разных зон. При нажатии этих экранных кнопок должен выполняться подсчет до 100 с приращениями в обработчике событий `setTimeout()`. Чтобы измерить время, которое для этого потребуется, воспользуйтесь глобальным контекстом, вызвав метод `performance.now()`, как показано ниже.

```
[src/app/app.component.ts]

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

@Component({
  selector: 'app-root',
```

```
  template: `
    <button (click)="runInsideAngularZone()">
      Run inside Angular zone
    </button>
    <button (click)="runOutsideAngularZone()">
      Run outside Angular zone
    </button>
  `
})
export class AppComponent {
  progress: number = 0;
  startTime: number = 0;

  constructor(private zone: NgZone) {}

  runInsideAngularZone() {
    this.start();
    this.step(() => this.finish('Inside Angular zone'));
  }

  runOutsideAngularZone() {
    this.start();
    this.step(() => this.finish('Outside Angular zone'));
  }

  start() {
    this.progress = 0;
    this.startTime = performance.now();
  }

  finish(location:string) {
    this.zone.run(() => {
      console.log(location);
      console.log('Took ' +
      (performance.now() - this.startTime) + 'ms');
    });
  }

  step(doneCallback: () => void) {
    if (++this.progress < 100) {
      setTimeout(() => {
        this.step(doneCallback);
      }, 10);
    } else {
    doneCallback();
  }
}
}
```

В данный момент обе экранные кнопки будут вести себя одинаково, поскольку их код выполняется в зоне Angular. Чтобы выполнить код одной

из них за пределами зоны Angular, необходимо вызвать метод runOutside Angular(), предоставляемый службой типа NgZone, как выделено ниже полужирным шрифтом.

```
runInsideAngularZone() {
  this.start();
  this.step(() => this.finish('Inside Angular zone'));
}
runOutsideAngularZone() {
  this.start();
  this.zone.runOutsideAngular(() => {
    this.step(() => this.finish('Outside Angular zone'));
  });
}
```

Теперь код одной из кнопок можно снова выполнять в зоне, а другой — за ее пределами, убедившись, что на его выполнение уходит (приблизительно) одинаковое время. И в этом нет ничего удивительного, поскольку обе кнопки по-прежнему выполняют одну и ту же задачу. Подключение библиотеки zone.js означает, что интерфейсы API браузера вставляются как прокладки за пределами Angular. И даже если выполнить код за пределами зоны Angular, это все равно будет означать его выполнение в самой зоне.

Чтобы увидеть отличия в производительности, необходимо внедрить какую-нибудь привязку данных в шаблон, как выделено ниже полужирным шрифтом.

```
[src/app/app.component.ts]

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

@Component({
  selector: 'app-root',
  template: `
    <h3>Progress: {{progress}}%</h3>
    <button (click)="runInsideAngularZone()">
      Run inside Angular zone
    </button>
    <button (click)="runOutsideAngularZone()">
      Run outside Angular zone
    </button>
  `
})
export class AppComponent {
  ...
}
```

Теперь должны быть заметны существенные отличия во времени выполнения кода в каждой кнопке. Данный пример наглядно показывает, что

издержки на привязку данных приводят к снижению производительности, тогда как метод `runOutsideAngular()` дает возможность добиться весьма существенных результатов в оптимизации производительности.

Принцип действия

Анализируя исходный код службы типа `NgZone`, можно обнаружить, что "внешняя" зона оказывается лишь самой верхней зоной в браузере. Каркас Angular разветвляет эту зону после инициализации и на ее основании строит изящную оболочку для службы типа `NgZone`.

Но поскольку зоны не разграничиваются в области асинхронных вызовов и привязки данных, то каждый вызов обработчика событий `setTimeout()` в зоне Angular распознается как событие, оказывающее влияние на процесс визуализации шаблона. При каждом его вызове в Angular обнаруживается обновление привязываемых данных вслед за асинхронной задачей, а далее повторно визуализируется представление. Когда это проделывается 100 раз, выполнение продлевается еще на несколько сотен миллисекунд.

Если выполнение данного кода происходит за пределами зоны Angular, каркасу Angular больше неизвестно о задачах, выполняемых обработчиком событий `setTimeout()`, а следовательно, повторная визуализация представления больше не требуется. Но по окончании завершающего шага итерации можно вызвать метод `NgZone.run()`, чтобы возвратить выполнение в зону Angular. Обнаружив задачу, Angular модифицирует данные и соответственно обновит их привязки. Но теперь это будет сделано лишь один раз.

Дополнение

В данном рецепте метод `run()` вызывается в методе `finish()` как в самой зоне Angular, так и за ее пределами. Если же метод `run()` вызывается в зоне, которая уже является контекстной зоной, где выполняется задача, то его вызов оказывается избыточным и по существу означает пустую операцию.

См. также

- Рецепт "Работа с зонами за пределами Angular" с пояснениями, каким образом зоны действуют в браузере.
- Рецепт "Прием событий от службы типа `NgZone`", где дается общее представление о том, как зоны применяются в Angular.
- Рецепт "Настройка компонентов на явное обнаружение изменений с помощью стратегии `OnPush`", в котором поясняется, как управлять вручную процессом обнаружения изменений в Angular.

Настройка компонентов на явное обнаружение изменений с помощью стратегии OnPush

Принятое в Angular соглашение, по которому поток данных проходит по иерархическому дереву компонентов сверху вниз, а события — снизу вверх, открывает ряд интересных возможностей. К их числу относится возможность контролировать момент, когда каркас Angular должен выполнить цикл обнаружения изменений в данном узле иерархического дерева компонентов.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/4909/**.

Подготовка

Начните со следующего простого приложения:

```
[app/root.component.ts]

import {Component} from '@angular/core';
import {Subject} from 'rxjs/Subject';
import {Observable} from 'rxjs/Observable';

@Component({
  selector: 'root',
  template: `
    <button (click)="shareSubject.next($event)">Share!</button>
    <article [shares]="shareEmitter"></article>
  `
})
export class RootComponent {
  shareSubject:Subject<Event> = new Subject();
  shareEmitter:Observable<Event> =
       this.shareSubject.asObservable();
}

[app/article.component.ts]

import {Component, Input, ngOnInit} from '@angular/core';
import {Observable} from 'rxjs/Observable';

@Component({
  selector: 'article',
  template: `
    <h1>{{title}}</h1>
    <p>Shares: {{count}}</p>
  `
})
export class ArticleComponent implements OnInit {
```

```
  @Input() shares:Observable<Event>;
  count:number = 0;
  title:string =
        'Insurance Fraud Grows In Wake of Apple Pie Hubbub';
  ngOnInit() {
    this.shares.subscribe((e:Event) => ++this.count);
  }
}
```

Это очень простое приложение, в котором наблюдаемый объект используется для передачи общих событий от родительского к порожденному компоненту. В порожденном компоненте ведется подсчет событий от щелчков кнопками мыши, а полученный результат помещается в элемент, расположенный на веб-странице. Цель состоит в том, чтобы внести в данное приложение такие коррективы, которые позволили бы обнаруживать в порожденном компоненте изменения лишь тогда, когда наблюдаемый объект отправляет соответствующее событие.

Реализация

В каркасе Angular уже имеется надежный способ обнаружения изменений с помощью зон. Всякий раз, когда в зоне наступает событие, каркас Angular распознает его как возможность для целенаправленной модификации приложения. И тогда он выполняет цикл обнаружения изменений, проходя иерархическое дерево компонентов сверху вниз и проверяя, требуются ли какие-нибудь обновления в изменившейся модели. А поскольку поток данных проходит по иерархическому дереву компонентов только сверху вниз, то данный способ уже является весьма эффективным для их обработки.

Но в данном случае может потребоваться дополнительное вмешательство, чтобы выполнять оптимизацию вручную, если обнаружение изменений должно происходить в самом компоненте. И тогда, исходя из того, что происходит вокруг компонента, нетрудно будет выяснить, когда он может измениться.

Настройка на стратегию обнаружения изменений

В каркасе Angular предоставляется возможность вносить коррективы в порядок обнаружения изменений. Так, если желательно отказаться от приема в Angular событий для запуска цикла обнаружения изменений, то компонент можно настроить на выполнение такого цикла только тогда, когда изменяются входные данные. Для этого достаточно настроить компонент на применение стратегии `OnPush` вместо выбираемой по умолчанию.

Безусловно, такое происходит намного реже, чем целый ряд событий в браузере. Если изменения в компоненте будут происходить только тогда,

когда изменятся входные данные, это избавит Angular от хлопот выполнять цикл обнаружения изменений в данном компоненте.

Итак, внесите в компонент `ArticleComponent` изменения, выделенные ниже полужирным шрифтом, чтобы применить в нем стратегию `OnPush`.

```
[app/article.component.ts]

import {Component, Input, ngOnInit, ChangeDetectionStrategy}
        from '@angular/core';
import {Observable} from 'rxjs/Observable';

@Component({
  selector: 'article',
  template: `
    <h1>{{title}}</h1>
    <p>Shares: {{count}}</p>
  `,
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class ArticleComponent implements OnInit {
  @Input() shares:Observable<Event>;
  count:number = 0;
  title:string =
        'Insurance Fraud Grows In Wake of Apple Pie Hubbub';
  ngOnInit() {
    this.shares.subscribe((e:Event) => ++this.count);
  }
}
```

В итоге стратегия обнаружения изменений успешно изменится, но при этом возникнет следующее существенное затруднение: подсчет событий больше не будет обновляться. И это, разумеется, происходит потому, что входные данные компонента не изменяются.

Подсчет событий обновлялся только потому, что прежде в Angular обнаруживались события от щелчков кнопками мыши, что приводило к обнаружению изменений в компоненте `ArticleComponent`. А теперь Angular предписано игнорировать эти события, несмотря на то, что подсчет событий в порожденном компоненте по-прежнему обновляется из обработчиков данных наблюдаемого объекта.

Запрашивание явного обнаружения изменений

Имеется также возможность внедрить ссылку на обнаружитель изменений в компоненте. По этой ссылке становится доступным метод, позволяющий принудительно начинать цикл обнаружения изменений всякий раз, когда в этом возникает потребность. А поскольку модель обновляется в обработчике данных наблюдаемого объекта, то такое место вполне пригодно для запуска

процесса обнаружения изменений, как показано ниже. Теперь вы увидите, что подсчет событий обновляется снова.

```
[app/article.component.ts]

import {Component, Input, ngOnInit, ChangeDetectionStrategy,
        ChangeDetectorRef} from '@angular/core';
import {Observable} from 'rxjs/Observable';

@Component({
  selector: 'article',
  template: `
    <h1>{{title}}</h1>
    <p>Shares: {{count}}</p>
  `,
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class ArticleComponent implements OnInit {
  @Input() shares:Observable<Event>;
  count:number = 0;
  title:string =
        'Insurance Fraud Grows In Wake of Apple Pie Hubbub';
  constructor(private changeDetectorRef_: ChangeDetectorRef) {}
  ngOnInit() {
    this.shares.subscribe((e:Event) => {
      ++this.count;
      this.changeDetectorRef_.markForCheck();
    });
  }
}
```

Принцип действия

Благодаря применяемой стратегии `OnPush` компонент, по существу, превращается в "черный ящик". Если входные данные не изменяются, то в Angular предполагается, что состояние в компоненте остается неизменным, а следовательно, нет никакой нужды продолжать обнаружение изменений дальше.

Обнаружение изменений всегда происходит в иерархическом дереве компонентов сверху вниз, и поэтому в запросе на обнаружение изменений вызывается метод `markForCheck()`, помечая обнаружитель изменений в компоненте для выполнения только в том случае, если Angular достигнет этого компонента в данном иерархическом дереве.

Дополнение

Данный рецепт оказывается полезным в тех случаях, когда требуется найти какой-нибудь способ выжать еще большую производительность из

приложения, но иногда такие действия могут привести не к полезному шаблону, а, наоборот, к антишаблону. Необходимость определять явным образом момент, когда в Angular следует выполнить цикл обнаружения изменений, может стать причиной дополнительных хлопот по мере роста объема исходного кода компонента. Так, в исходном коде компонента могут возникнуть программные ошибки из-за пропуска мест, где метод `markForCheck()` так и не был вызван. Принятая в Angular стратегия обнаружения изменений уже достаточно производительна и надежна, поэтому рассмотренной здесь настройкой на другую стратегию следует пользоваться благоразумно.

См. также

- Рецепт "Работа с зонами за пределами Angular" с пояснениями, каким образом зоны действуют в браузере.
- Рецепт "Прием событий от службы типа `NgZone`", где дается общее представление о том, как зоны применяются в Angular.
- Рецепт "Выполнение за пределами зоны Angular", в котором показывается, как выполнять долгосрочные операции, не требуя дополнительных издержек на зоны.
- Рецепт "Настройка инкапсуляции представлений на максимальную производительность", где показывается, как настраивать компоненты на применение теневой модели DOM.

Настройка инкапсуляции представлений на максимальную производительность

Как бы это ни казалось банальным, но версия Angular 2 рассчитана на браузеры будущего. Такому утверждению можно найти немало веских оснований, но особенно справедливым оно в отношении инкапсуляции представлений.

Идеальной для Angular 2 является такая модель компонентов, в которой компоненты полностью изолированы в предметной области, кроме нескольких частей, внешне доступных для модификации. В этом отношении модель компонентов вполне справляется со своей задачей, но даже самые современные браузеры ограничивают ее стремление к подобной эффективности. И это особенно справедливо в отношении стилевого оформления с помощью правил CSS.

В связи с изложенным выше особое значение приобретают следующие возможности стилевого оформления в Angular.

- Создание стилей, которые гарантированно должны применяться только в данном компоненте.
- Явное указание стилей, которые должны наследоваться по иерархии в дереве компонентов.
- Задание стратегии инкапсуляции по частям.

Имеется немало интересных способов получить эффективную схему стилевого оформления компонентов.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/1463/**.

Подготовка

Начните со следующего простого приложения:

```
[src/app/app.component.ts]

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

@Component({
  selector: 'app-root',
  template: `
    <h1>
      {{title}}
    </h1>
  `,
  styles: [`
    h1 {
      color: red;
    }
  `]
})
export class AppComponent {
  title = 'app works!';
}
```

Реализация

По умолчанию теневая модель по спецификации Shadow DOM не применяется в Angular для компонентов, поскольку в большинстве браузеров она не поддерживается на удовлетворительном уровне. В качестве наилучшей ее альтернативы по умолчанию выбирается эмуляция подобной инкапсуляции стилевого оформления путем вложения соответствующих селекторов. Приведенный выше компонент, по существу, равнозначен следующему:

```
[src/app/app.component.ts]

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

@Component({
  selector: 'app-root',
  template: `
    <h1>
      {{title}}
    </h1>
  `,
  styles: [`
    h1 {
      color: red;
    }
  `],
  encapsulation: ViewEncapsulation.Emulated
})
export class AppComponent {
  title = 'app works!';
}
```

Эмулируемая инкапсуляция стилевого оформления

Каким же образом такая эмуляция выполняется в Angular? Чтобы выяснить этот вопрос, проанализируйте визуализированное приложение. Ваш компонент должен принять следующий вид:

```
<app-root _nghost-mqf-1="">
  <h1 _ngcontent-mqf-1="">
    app works!
  </h1>
</app-root>
```

А в заголовке документа появится приведенное ниже стилевое оформление:

```
<style>
  h1[_ngcontent-mqf-1] {
    color: red;
  }
</style>
```

Теперь общая картина должна немного проясниться. В частности, Angular присваивает классу, но не экземпляру данного компонента однозначный идентификатор, а стили, определенные для глобального документа, будут применены только в тех дескрипторах, где имеется соответствующий атрибут.

Отмена инкапсуляции стилевого оформления

Если такая инкапсуляция не требуется, то можно отменить ее следующим образом:

```
encapsulation: ViewEncapsulation.None
```

Стадия присваивания однозначного идентификатора в этом случае будет благополучно пропущена, что в итоге приведет к следующей привычной разметке документа:

```
<app-root>
  <h1>
    app works!
  </h1>
</app-root>
```

А в заголовке документа появится приведенное ниже стилевое оформление.

```
<style>
  h1 {
    color: red;
  }
</style>
```

Естественная инкапсуляция стилевого оформления

Самый лучший, передовой и наименее поддерживаемый способ инкапсуляции стилевого оформления состоит в том, чтобы воспользоваться экземплярами теневой модели DOM вместе с каждым экземпляром компонента. И это можно сделать, установив вид естественной (native) инкапсуляции следующим образом:

```
encapsulation: ViewEncapsulation.Native
```

В итоге компонент будет визуализирован так, как показано ниже.

```
<app-root>
#shadow-root
<style>
  h1 {
    color: red;
  }
</style>
  <h1>
    app works!
  </h1>
</app-root>
```

Принцип действия

Каркас Angular обладает достаточно развитой логикой, чтобы выяснить, где требуется разместить стили и как сделать их пригодными для конфигурации приложения. Ниже перечислены виды инкапсуляции представления, устанавливаемые в Angular.

- **`None`** и **`Emulated`** — стили размещаются в заголовке документа.
- **`Native`** — стили сопутствуют визуализируемому компоненту.
- **`None`** и **`Native`** — никой модификации стилей не требуется.
- **`Emulated`** — стили ограничиваются селекторами атрибутов.

Дополнение

При выборе видов инкапсуляции представления следует принимать во внимание эффективность стилевого оформления с помощью правил CSS. Хорошо известно и вполне логично, то такое стилевое оформление оказывается более эффективным, когда для этого требуется обойти лишь часть модели DOM и использовать для сопоставления более простые селекторы.

Эмуляция инкапсуляции стилевого оформления добавляет селектор атрибутов буквально в каждый стиль, определенный в данном компоненте. При масштабировании нетрудно обнаружить, насколько это может снизить производительность. Теневая модель DOM изящно устраняет подобное затруднение, предоставляя немодифицированные стили в ограниченной части данной модели. Ее стили нельзя изолировать, но можно применить к другим компонентам вниз по иерархии. Кроме того, компоненты теневой модели DOM допускают вложение и стратегическое применение.

См. также

- Рецепт "Общее представление о функции `enableProdMode()` и надлежащем ее применении в чистых и нечистых каналах" с пояснениями, как придать приложению рабочую форму.
- Рецепт "Настройка компонентов на явное обнаружение изменений с помощью стратегии `OnPush`" с пояснениями, как управлять вручную процессом обнаружения изменений в Angular.

Настройка службы типа `Renderer` на применение рабочих веб-процессов

К числу наиболее привлекательных нововведений в новую версию каркаса Angular относится полная абстракция процесса визуализации. В основу такой абстракции положен один из главных принципов Angular: возможность

незаметно подставлять один модуль вместо другого, изменяя поведение. И это, безусловно, означает, что в Angular не допускаются зависимости, выходящие за пределы модулей.

Особое внимание в отношении конфигурирования Angular уделяется месту, где происходит выполнение кода. Это проявляется по-разному, и в данном рецепте делается акцент на способности Angular выполнять визуализацию не в том месте, где обычно действует исполняющая среда JavaScript в основных браузерах.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/1859/**.

Подготовка

Начните со следующих простых элементов разметки, которые пока еще не образуют полноценное приложение:

```
[index.html]

<!DOCTYPE html>
<html>
  <head>
    <script src="zone.js "></script>
    <script src="reflect-metadata.js"></script>
    <script src="system.src.js"></script>
    <script src="system-config.js"></script>
  </head>
  <body>
    <article></article>
      <script>
        System.import('system-config.js')
              .then(function() {
                System.import('main.ts');
              });
      </script>
  </body>
</html>

[app/article.component.ts]

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

@Component({
  selector: 'article',
  template: `
    <h2>{{title}}</h2>
  `
})
```

```
export class ArticleComponent {
  title:string = 'Survey Indicates Plastic Funnel
                 ↳Best Way to Drink Rare Wine';
}
```

Обратите внимание на то, что в данном рецепте для работы с модулями транспиляции исходного кода TypeScript применяется библиотека SystemJS, хотя это делается ради простоты демонстрируемого примера. Для достижения того же самого результата в скомпилированном надлежащим образом приложении можно воспользоваться обычными языковыми средствами JavaScript, исключив зависимость от библиотеки SystemJS.

Данный рецепт начинается, исходя из предположения, что вам в основном известно, что такое рабочие веб-процессы и как они действуют. Подробнее об их особенностях речь пойдет далее в этом рецепте.

Реализация

В соответствии с конфигурацией запуска, устанавливаемой в библиотеке SystemJS, рассматриваемое здесь приложение запускается на выполнение из исходного файла `main.ts`, и поэтому начинать следует именно с него. Вместо обычной начальной загрузки приложения из этого файла воспользуйтесь импортированной вспомогательной функцией Angular, чтобы инициализировать экземпляр рабочего веб-процесса, как показано ниже.

```
[main.ts]

import {bootstrapWorkerUi} from "@angular/platform-webworker";

bootstrapWorkerUi(window.location.href + "loader.js");
```

Не вдаваясь особенно в подробности того, что при этом происходит, достаточно сказать, что данная функция создает экземпляр рабочего веб-процесса, интегрированный со службой типа `Renderer` в Angular.

Напомним, что для инициализации рабочего веб-процесса требуется указать путь к файлу его запуска на JavaScript, а относительный путь в этом файле может не всегда сработать. Именно поэтому местоположение окна используется здесь для предоставления абсолютного URL. Потребность в этом может зависеть от конкретной настройки вашей среды разработки.

В рассматриваемом здесь файле делается ссылка на файл инициализации рабочего веб-процесса под названием `loader.js`, и поэтому создайте его, как показано ниже. Для рабочих веб-процессов нельзя назначить исходный файл TypeScript.

```
[loader.js]
```

```
importScripts(
  "system.js",
  "zone.js",
  "reflect-metadata.js",
  "./system-config.js");

System.import("./web-worker-main.ts");
```

В этом файле сначала импортируются те же самые файлы, которые перечислены в дескрипторе <head> из файла index.html. И в этом должен быть определенный смысл, поскольку на рабочий веб-процесс в Angular возлагается ряд обязанностей, хотя и без прямого доступа ко всему, что существует в главной исполняющей среде JavaScript. Например, данный рабочий веб-процесс будет визуализировать компонент, и поэтому он должен быть в состоянии правильно интерпретировать обозначение @Component({}), чего нельзя сделать без расширения reflect-metadata.

Как и у основного приложения, у рабочего веб-процесса имеется свой файл инициализации web-worker-main.ts, принимающий следующую форму:

```
[web-worker-main.ts]

import {AppModule} from './app/app.module';
import {platformWorkerAppDynamic}
        from '@angular/platform-webworker-dynamic';

platformWorkerAppDynamic().bootstrapModule(AppModule);
```

По сравнению с обычным файлом main.ts, этот файл выглядит очень знакомым. В каркасе Angular предоставляется совершенно отдельный модуль платформы, хотя и с таким же интерфейсом API для начальной загрузки приложения из еще не определенного модуля AppModule. Поэтому определите его следующим образом:

```
[app/app.module.ts]

import {NgModule} from '@angular/core';
import {WorkerAppModule} from '@angular/platform-webworker';
import {AppModule} from './app.module';
import {ArticleComponent} from './article.component';

@NgModule({
  imports: [
    WorkerAppModule
  ],
  bootstrap: [
    ArticleComponent
  ],
```

```
  declarations: [
    ArticleComponent
  ]
})
export class AppModule {}
```

Подобно модулю `BrowserModule`, обычно применяемому в стандартном модуле приложения верхнего уровня, для рабочих веб-процессов в Angular предоставляется модуль `WorkerAppModule`, обеспечивающий всю необходимую их интеграцию.

Принцип действия

Однако не следует ошибочно думать, что приложение теперь работает в двух отдельных потоках исполнения JavaScript. На самом деле рабочие веб-процессы представляют собой всего лишь глухие блоки исполнения кода JavaScript. После инициализации они получают для исполнения первоначальный фрагмент кода JavaScript, что в данном рецепте принято в форме файла `loader.js`. Они не имеют ни малейшего представления, что происходит в главной исполняющей среде браузера, и не могут взаимодействовать с моделью DOM, поэтому обмен данными с ними возможен только сообщениями через интерфейс `postMessage`. Над этим интерфейсом в Angular строится изящная абстракция для создания шинного интерфейса, соединяющего обе исполняющие среды вместе.

Если проанализировать спецификацию интерфейса `postMessage`, то можно заметить, что все данные, передаваемые в виде сообщений, должны быть сериализированы в символьную строку. Как же тогда данная конфигурация визуализации может вообще взаимодействовать с моделью DOM в главной исполняющей среде браузера, обрабатывая события и отображая HTML-разметку, а рабочий веб-процесс — визуализировать то, что ему недоступно в модели DOM?

Ответ на этот вопрос прост: Angular сериализирует все в последовательную форму. Когда происходит первоначальная визуализация или наступает событие в браузере, каркас Angular извлекает все, что ему требуется знать о текущем состоянии, заключает в сериализированную символьную строку и отправляет средству визуализации в рабочем веб-процессе по соответствующему каналу. А это средство правильно интерпретирует то, что ему передается. И хотя ему недоступна основная структура модели DOM, оно, безусловно, способно создать элементы HTML-разметки, правильно интерпретировать события, передаваемые ему в последовательной форме, чтобы обработать их и визуализировать.

Когда же наступит момент сообщить браузеру, что именно он должен отобразить, средство визуализации в рабочем веб-процессе, в свою очередь,

сериализирует визуализированный компонент и отправит его обратно в главную исполняющую среду браузера, где полученная символьная строка будет распакована и вставлена в модель DOM.

Все описанное выше представляется каркасу Angular обычным, поскольку он абстрагируется от всех зависимостей, способных этому помешать в браузере. В частности, события поступают и обрабатываются своим чередом, средство визуализации сообщает, что именно следует разместить в модели DOM, а все, что происходит между тем, совершенно не важно для каркаса Angular. Его вообще не интересует, что происходит в полностью обособленной исполняющей среде JavaScript.

Обратите внимание на то, что элементы, с которых был начат данный рецепт, не содержат ничего такого, что допускало бы совместимость с рабочими веб-процессами. Именно в этом и состоит все изящество абстракции рабочих веб-процессов в Angular.

Дополнение

По мере совершенствования поддержки и реализации рабочих веб-процессов шаблоны, подобные рассмотренному выше, могут найти все большее распространение, и в этом отношении разработчики Angular проявили завидную прозорливость. Тем не менее во внимание необходимо принять ряд важных соображений.

Оптимизация ради повышения производительности

Одно из главных преимуществ, которые дает применение рабочих веб-процессов, заключается в наличии доступа к контексту исполнения, который ничем не блокируется в контексте исполнения браузера. Что касается таких факторов снижения производительности, как перекомпоновка документа и блокировка обработчиков в цикле ожидания событий, то рабочий веб-процесс продолжит свое выполнение, не обращая никакого внимания на происходящее где-нибудь в другом месте.

Следовательно, повысить производительность благодаря оптимизации становится затруднительно. Для обмена данными между двумя потоками исполнения JavaScript требуется сериализация и передача событий, что, очевидно, делается не так быстро, как при их обработке в одном и том же потоке исполнения. А в особенно сложном приложении визуализация может быстро стать одной из самых затратных операций. Поэтому вам придется поэкспериментировать, чтобы принять взвешенное решение в отношении производительности вашего приложения, поскольку применение рабочих веб-процессов приводит к повышению производительности только в тех приложениях, где визуализация становится непозволительно затратной операцией.

Соображения по поводу совместимости

Рабочие веб-процессы находят очень хорошую поддержку, но все еще имеется немало браузеров, где они вообще не поддерживаются. Если ваше приложение рассчитано на универсальное применение, то применять в нем рабочие веб-процессы не рекомендуется, поскольку они некорректно выходят из сбойной ситуации, что может привести приложение к полному отказу.

См. также

- Рецепт "Настройка службы типа `Renderer` на применение рабочих веб-процессов", где поясняется весь процесс настройки приложения для визуализации в рабочем веб-процессе.
- Рецепт "Настройка приложений на компиляцию перед выполнением" с пояснениями, как скомпилировать приложение во время сборки.
- Рецепт "Настройка приложений на загрузку по требованию", в котором показывается, как задерживать выполнение отдельных фрагментов кода до тех пор, пока они не понадобятся.

Настройка приложений на компиляцию перед выполнением

В версии Angular 2 внедрена концепция *компиляции перед выполнением* (ahead-of-time compilation, или AOT), иначе называемая *статической компиляцией*. Это другой вариант компиляции, отличающейся от компиляции программы прямо в браузере во время выполнения программы (она называется *динамической компиляцией*, Just-in-time compilation, или JIT) . Она выполняется на сервере перед запуском программы.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **`http://ngcookbook.herokuapp.com/9253/`**.

Подготовка

Компиляция перед выполнением не зависит от конкретного приложения, и поэтому ее можно использовать в любом существующем приложении Angular 2, внеся в него минимальные изменения. Для целей данного рецепта допустим, что в файле `app/app.module.ts` находится существующий модуль `AppModule`. Его содержимое не имеет в данном случае особого значения, поскольку оно никак не связано с целью компиляции перед выполнением.

Реализация

Применение компиляции перед выполнением означает, что компиляция и начальная загрузка приложения будут происходить по-разному. В зависимости от конкретной конфигурации имена родственных файлов, возможно, придется снабдить суффиксом **aot**. Хотя это делается лишь в организационных целях, поскольку порядок именования файлов в Angular не имеет никакого значения.

Установка зависимостей от компиляции перед выполнением

Для этой цели потребуется новое инструментальное средство компиляции `ngc`, входящее в состав пакета `@angular/compiler-cli`. Кроме того, для начальной загрузки служит функция `platformBrowser()`, предоставляемая в пакете `@angular/platform-browser`. Поэтому установите их и сохраните зависимость от них в файле `package.json` по следующей команде:

```
npm install @angular/compiler-cli @angular/platform-server --save
```

Настройка компилятора `ngc`

Компилятор `ngc` выполняет подмножество обязанностей обычного компилятора `tsc`. Поэтому его целесообразно снабдить отдельным файлом конфигурации под названием `tsconfig-aot.json`, хотя оно может быть иным. Этот файл должен быть подобным уже существующему файлу конфигурации `tsconfig.json`, но в него необходимо внести следующие важные изменения:

```
[tsconfig-aot.json]

{
  "compilerOptions": {
    (здесь может быть куча настроек, измените только параметр 'module')
    "module": "es2015",
  },
  "files": [
    "app/app.module.ts",
    "main.ts"
  ],
    "angularCompilerOptions": {
    "genDir": "aot",
    "skipMetadataEmit" : true
  }
}
```

Приведение определений компонентов в соответствие с требованиями компиляции перед выполнением

Для компиляции перед выполнением приложение требуется организовать особым образом, приняв соответствующие меры. В частности, ничто не должно нарушить уже существующее приложение, но эти меры важны и заслуживают особого внимания. Принять их нетрудно, если следовать нормам передовой практики в Angular.

Прежде всего, в определениях модулей должен быть задан параметр `moduleId`. Как правило, в компонентах, формируемых с помощью интерфейса Angular CLI, этот параметр уже задан. В противном случае в нем следует указать значение `module.id`, как показано ниже.

```
@Component({
  moduleId: module.id,
  (остальные параметры)
})
```

В процессе компиляции перед выполнением модуль оказывается неопределенным, поэтому в корневом шаблоне ему можно присвоить фиктивное значение следующим образом:

```
<script>window.module = 'aot';</script>
```

И хотя для компиляции перед выполнением идентификатор модуля не требуется, он позволяет произвести компиляцию без ошибок. Однако рассматриваемые здесь меры по установке и настройке параметра **moduleId** могут быть изменены или вообще исключены в последующих версиях Angular, поскольку они существуют лишь для того, чтобы обеспечить совместимость компиляции перед выполнением и динамической компиляции.

Далее необходимо изменить путь к шаблонам, как CSS, так и HTML, указав его относительно файла определения компонента, а не корневого компонента приложения. Если вы пользуетесь соглашением, принятым в интерфейсе Angular CLI, или руководством по стилевому оформлению, то, скорее всего, уже сделали это.

Компиляция средствами `ngc`

Компилятор `ngc` нельзя вызвать из командной строки, и поэтому его двоичный файл придется запустить непосредственно. А поскольку в данном рецепте не используется стандартное соглашение об именах, принятое для файла конфигурации `tsconfig.json`, вам придется указать для двоичного файла компилятора место нахождения альтернативного файла конфигурации. Для того чтобы выполнить компиляцию, запустите следующую команду из корневого каталога вашего приложения:

```
node_modules/.bin/ngc -p tsconfig-aot.json
```

По этой команде будет получена коллекция экземпляров типа NgFactory в файлах с расширением **.ngfactory.ts**, размещаемых в каталоге aot, как было указано ранее в разделе angularCompilerOptions файла конфигурации tsconfig-aot.json. Но по завершении компиляции в приложении все еще остается неизвестно, как воспользоваться полученными экземплярами типа NgFactory.

Начальная загрузка с компиляцией перед выполнением

Теперь ваше приложение начнет свою работу с экземпляра типа AppModuleNgFactory вместо экземпляра типа AppModule. Организуйте его начальную загрузку с помощью функции platformBrowser(), как показано ниже. В итоге вы сможете запустить свое приложение обычным образом, но на этот раз, используя предварительно скомпилированные файлы.

```
[main.ts]

import {platformBrowser} from '@angular/platform-browser';
import {AppModuleNgFactory} from 'aot/app/app.module.ngfactory';

platformBrowser().bootstrapModuleFactory(AppModuleNgFactory);
```

Принцип действия

Компиляция в Angular — дело непростое, тем не менее, можно указать на ряд следующих основных моментов, касающихся перехода приложения к процессу компиляции перед выполнением:

- Компилятор Angular находится в довольно крупном модуле @angular/compiler. А применение компиляции перед выполнением означает, что данный модуль больше не требуется передавать в браузер, что позволит существенно уменьшить нагрузки на каналы связи.
- Процесс компиляции в Angular включает в себя извлечение из компонентов символьных строк шаблона, указанных в свойствах template или templateUrl, и последующее их преобразование в экземпляры типа NgFactory. Эти экземпляры устанавливают порядок интерпретации шаблона в Angular. Преобразование в экземпляры типа Factory происходит как при динамической компиляции, так и при компиляции перед выполнением. А переход к компиляции перед выполнением означает, что подобная обработка выполняется на стороне сервера, а не клиента, для чего непосредственно используются определения из экземпляров типа NgFactory вместо нескомпилированных символьных строк шаблона.

- Очевидно, что компиляция перед выполнением замедляет процесс сборки. Ведь всякий раз, когда приходится обновлять кодовую базу, требуются дополнительные вычисления. А поскольку правильность интерпретации в Angular определений из экземпляров типа `NgFactory`, формируемых при компиляции перед выполнением, не вызывает особых сомнений, то динамическую компиляцию лучше производить на стадии разработки приложений, а компиляцию перед выполнением — на стадии сборки и развертывания рабочих приложений.

Дополнение

Компиляция перед выполнением замечательна, но может быть пригодна не для всех. Она, скорее всего, позволит сократить время загрузки и инициализации приложения Angular, но в то же время значительно усложнит процесс сборки. Более того, если будет решено применять динамическую компиляцию на стадии разработки, а компиляцию перед выполнением — на стадии подготовки к эксплуатации, то придется сопровождать две версии трех разных файлов: `index.html`, `main.ts` и `tsconfig.json`. Возможно, эти дополнительные издержки на стадии сборки и оправданны, тем не менее, они требуют взвешенного решения исходя из конкретной ситуации, сложившейся на стадии разработки.

Дополнительная оптимизация на стадии встряхивания дерева

В каркасе Angular поддерживается также отдельная стадия оптимизации, называемая *встряхиванием дерева* (*tree shaking*). На этой стадии применяется отдельная библиотека `npm` под названием `rollup`. По существу, эта библиотека читает весь прикладной код (в виде исходных файлов JavaScript), выявляет неиспользуемые модули и удаляет их из скомпилированной кодовой базы, а следовательно, из полезной информации, доставляемой в браузер. Это аналогично встряхиванию дерева с целью избавиться от омертвевших листьев, которые при этом опадают на землю. Отсюда и название данной стадии оптимизации прикладного кода.

Конфигурация модуля **es2015**, указанная ранее в файле **tsconfig-aot.json**, была нацелена на поддержку библиотеки **rollup**, поскольку это требуется ради совместимости.

Если сопровождение кодовой базы вашего приложения организовано надлежащим образом, то вам вряд ли удастся извлечь большие выгоды из стадии встряхивания дерева, поскольку неиспользуемые импортируемые модули и прочие компоненты будут исключены на стадии программирования. Более того, можно было бы привести веские аргументы в пользу того, что встряхивание дерева способно превратиться в антишаблон, поскольку оно

невольно побуждает либеральных разработчиков включать модули в прикладной код, зная, что на стадии встряхивания дерева эти модули все равно будут исключены как неиспользуемые в приложении. А это может привести к загромождению кодовой базы. Встряхивание дерева может оказаться удобным средством оптимизации прикладного кода в Angular 2, но пользы от него практически никакой, если содержать кодовую базу в опрятном виде.

См. также

- Рецепт "Общее представление о функции `enableProdMode()` и надлежащем ее применении в чистых и нечистых каналах", где поясняется, как придать приложению рабочую форму.
- Рецепт "Настройка службы типа `Renderer` на применение рабочих веб-процессов", в котором поясняется весь процесс настройки приложения для визуализации в рабочем веб-процессе.
- Рецепт "Настройка приложений на загрузку по требованию", в котором показывается, как задерживать выполнение отдельных фрагментов кода до тех пор, пока они не понадобятся.

Настройка приложений на загрузку по требованию

По требованию загружаются приложения, в которых извлечение соответствующих ресурсов откладывается до тех пор, пока в них фактически не возникнет потребность. Как только начнется масштабирование приложений, загрузка по требованию может принести ощутимый выигрыш в производительности. И в версии Angular 2 она поддерживается исходно.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/0279**.

Подготовка

Начните со следующего простого приложения:

```
[app/root.component.ts]

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

@Component({
  selector: 'root',
  template: `
    <h1>Root component</h1>
    <router-outlet></router-outlet>
```

```
  `
})
export class RootComponent {}

[app/link.component.ts]

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

@Component({
  selector: 'app-link',
  template: `
    <a routerLink="/article">article</a>
  `
})
export class LinkComponent {}

[app/article.component.ts]

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

@Component({
  selector: 'article',
  template: `
    <h1>{{title}}</h1>
  `
})
export class ArticleComponent {
  title:string =
        'Baboon's Stock Picks Crush Top-Performing Hedge Fund';
}

[app/app.module.ts]

import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {RouterModule, Routes} from '@angular/router';
import {RootComponent} from './root.component';
import {ArticleComponent} from './article.component';
import {LinkComponent} from './link.component';

const appRoutes:Routes = [
  {
    path: 'article',
    component: ArticleComponent
  },
  {
    path: '**',
    component: LinkComponent
  }
];
```

```
@NgModule({
  imports: [
    BrowserModule,
    RouterModule.forRoot(appRoutes)
  ],
  declarations: [
    ArticleComponent,
    LinkComponent,
    RootComponent
  ],
  bootstrap: [
    RootComponent
  ]
})
export class AppModule {}
```

В этом приложении имеются только два маршрута: стандартный маршрут с отображением ссылки на страницу со статьей, а также маршрут статьи с отображением компонента `ArticleComponent`. Цель состоит в том, чтобы отложить загрузку ресурсов, требующихся для маршрута статьи, вплоть до ее фактического посещения.

Реализация

Загрузка по требованию означает, что у первоначально загружаемого модуля приложения не должно быть никаких зависимостей от того модуля, который требуется загрузить по требованию, поскольку его исходный код не появится прежде посещения ресурса по новому URL. Поэтому перенесите сначала ссылку на компонент `ArticleComponent` в его собственный модуль, как показано ниже.

```
[app/article.module.ts]

import {NgModule} from '@angular/core';
import {ArticleComponent} from './article.component';

@NgModule({
  declarations: [
    ArticleComponent
  ],
  exports: [
    ArticleComponent
  ]
})
export class ArticleModule {}
```

Удаление всех зависимостей означает также перемещение определений соответствующих маршрутов в данный модуль, как выделено ниже полужирным шрифтом.

[app/article.module.ts]

```
import {NgModule} from '@angular/core';
import {ArticleComponent} from './article.component';
import {Routes, RouterModule} from '@angular/router';

const articleRoutes:Routes = [
  {
    path: '',
    component: ArticleComponent
  }
];

@NgModule({
  imports: [
    RouterModule.forChild(articleRoutes)
  ],
  declarations: [
    ArticleComponent
  ],
  exports: [
    ArticleComponent
  ]
})
export class ArticleModule {}
```

Далее удалите все ссылки на эти модули из модуля AppModule. Кроме того, внесите изменение в определение маршрута, чтобы просто ссылаться на путь к модулю, загружаемому по требованию, а не указывать компонент для визуализации. Откорректируйте также имя модуля, используя специальный синтаксис **#**, как показано ниже.

[app/app.module.ts]

```
import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {RouterModule, Routes} from '@angular/router';
import {RootComponent} from './root.component';
import {LinkComponent} from './link.component';

const appRoutes:Routes = [
  {
    path: 'article',
    loadChildren: './app/article.module#ArticleModule'
  },
  {
    path: '**',
    component: LinkComponent
  }
```

```
];
@NgModule({
  imports: [
    BrowserModule,
    RouterModule.forRoot(appRoutes)
  ],
  declarations: [
    LinkComponent,
    RootComponent
  ],
  bootstrap: [
    RootComponent
  ]
})
export class AppModule {}
```

Вот, собственно, и все, что требуется для организации загрузки по требованию. Теперь ваше приложение должно вести себя таким же образом, как и в начале данного рецепта.

Принцип действия

Чтобы убедиться, что загрузка действительно выполняется по требованию, запустите свое приложение на выполнение и внимательно присмотритесь к вкладке Network (Сеть) на консоли разработчика вашего браузера. Щелкнув на ссылке `routerLink`, ведущей к статье, вы должны заметить, что запросы файлов `article.module.ts` и `article.component.ts` отправляются раньше, чем происходит визуализация. Это означает, что требующиеся в Angular файлы извлекаются только после фактического посещения ресурса по выбранному маршруту. В свойстве маршрута `loadChildren` для Angular задается следующее условие: если модуль еще не загружен, то при посещении ресурса по данному маршруту необходимо использовать относительный путь, который мог быть предоставлен для извлечения модуля. И как только файл модуля будет извлечен, Angular сможет синтаксически проанализировать его, чтобы выяснить, какие еще файлы потребуются для запроса загрузки по всем зависимостям данного модуля.

Дополнение

Обратите внимание на то, что загрузка по требованию вносит дополнительную задержку в приложение, поскольку Angular приходится ждать загрузки ресурсов именно тогда, когда они фактические нужны. Более того, в примере из данного рецепта выполняются *два* полных обращения к серверу при посещении статьи по заданному URL: одно — для запроса файла модуля, а другое — для запроса ресурсов, от которых зависит данный модуль.

В рабочей среде такая задержка может оказаться неприемлемой. В качестве выхода из этого положения можно скомпилировать загружаемую по требованию полезную информацию в один файл, который может быть извлечен по запросу. Хотя такая мера может дать разные результаты в зависимости от того, каким образом построено приложение.

Учет общих модулей

Загружаемый по требованию модуль полностью обособлен от главного модуля приложения, а это подразумевает внедряемые зависимости. Так, если предоставить службу для модуля приложения верхнего уровня, то можно обнаружить, что в нем будут созданы два отдельных экземпляра этой службы на каждом месте, где она предоставляется. И это, безусловно, неожиданное поведение, если учесть, что в нормально загружаемом приложении будет создан лишь один экземпляр службы, если только она предоставлена.

В качестве выхода из положения можно употребить метод `forRoot()`, применяемый в Angular для одновременного предоставления и конфигурирования служб. В данном рецепте он позволяет формально предоставить службу в нескольких местах, но в Angular ее дубликаты будут проигнорированы при условии, что это делается в теле метода `forRoot()`.

Прежде всего определите службу типа `AuthService`, которую требуется предоставить лишь в одном экземпляре. В приведенное ниже определение данной службы включен оператор протоколирования, чтобы можно было увидеть получение лишь одного ее экземпляра.

```
[app/auth.service.ts]

import {Injectable} from '@angular/core';

@Injectable()
export class AuthService {
  constructor() {
    console.log('instantiated AuthService');
  }
}
```

Далее создайте следующую оболочку типа `NgModule` специально для данной службы:

```
[app/auth.module.ts]

import {NgModule, ModuleWithProviders} from "@angular/core";
import {AuthService} from "./auth.service";

@NgModule({})
export class AuthModule {
  static forRoot():ModuleWithProviders {
```

```
    return {
      ngModule: AuthModule,
      providers: [
        AuthService
      ]
    };
  }
}
```

Как следует из приведенного выше кода, здесь применяется стратегия, основанная на вызове метода `forRoot()`. Поэтому модуль `AuthModule` можно свободно импортировать как в самом модуле приложения, так и по требованию:

```
[app/app.module.ts]

import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {RouterModule, Routes} from '@angular/router';
import {RootComponent} from './root.component';
import {LinkComponent} from './link.component';
import {AuthModule} from './auth.module';

const appRoutes:Routes = [
  {
    path: 'article',
    loadChildren: './app/article.module#ArticleModule'
  },
  {
    path: '**',
    component: LinkComponent
  }
];

@NgModule({
  imports: [
    BrowserModule,
    RouterModule.forRoot(appRoutes),
    AuthModule.forRoot()
  ],
  declarations: [
    LinkComponent,
    RootComponent
  ],
  bootstrap: [
    RootComponent
  ]
})
export class AppModule {}
```

Добавьте данный модуль и в загружаемый по требованию модуль, но только не вызывайте метод `forRoot()`, как выделено ниже полужирным шрифтом. Этот метод зарезервирован только для корневого модуля приложения.

[app/article.module.ts]

```
import {NgModule} from '@angular/core';
import {ArticleComponent} from './article.component';
import {Routes, RouterModule} from '@angular/router';
import {AuthModule} from './auth.module';

const articleRoutes:Routes = [
  {
    path: '',
    component: ArticleComponent
  }
];

@NgModule({
  imports: [
    RouterModule.forChild(articleRoutes),
    AuthModule
  ],
  declarations: [
    ArticleComponent
  ],
  exports: [
    ArticleComponent
  ]
})
export class ArticleModule {}
```

И, наконец, внедрите службу в компоненты `RootComponent` и `ArticleComponent` и воспользуйтесь операторами протоколирования, чтобы убедиться, что она действительно достигает обоих компонентов, как показано ниже. В итоге вы должны увидеть, как получается единственный экземпляр службы, которая успешно внедряется в оба компонента.

[app/root.component.ts]

```
import {Component} from '@angular/core';
import {AuthService} from './auth.service';

@Component({
  selector: 'root',
  template: `
    <h1>Root component</h1>
    <router-outlet></router-outlet>
  `
})
```

```
export class RootComponent {
  constructor(private authService_:AuthService) {
    console.log(authService_);
  }
}

[app/article.component.ts]

import {Component} from '@angular/core';
import {AuthService} from './auth.service';

@Component({
  selector: 'article',
  template: `
    <h1>{{title}}</h1>
  `
})
export class ArticleComponent {
  title:string =
        'Baboon's Stock Picks Crush Top-Performing Hedge Fund';

  constructor(private authService_:AuthService) {
    console.log(authService_);
  }
}
```

См. также

- Рецепт "Настройка службы типа `Renderer` на применение рабочих веб-процессов", в котором поясняется весь процесс настройки приложения для визуализации в рабочем веб-процессе.
- Рецепт "Настройка приложений на компиляцию перед выполнением", в котором поясняется, как скомпилировать приложение во время сборки.

Предметный указатель

Р

Promises 160
Protractor 367

А

Ад обратных вызовов,
    проявление 165

Б

Библиотеки
  ReactiveX
    возможности 189
    реализация 188
  reflectmetadata, назначение и
      применение 320
  rollup, применение 436
  RxJS
    метод .toPromise(), применение 184
    модули, применение 188
    поддержка наблюдаемых
      объектов 181
    реализация и назначение 188
    шаблон 'Наблюдатель',
      реализация 198
  SystemJS
    конфигурирование 331
    назначение 326
    применение 428
    транспиляция кода TypeScript 325
  zone.js, применение 401, 409, 410

В

Внедрение зависимостей
  система
    в Angular 2 280
    сопоставления зависимостей 295
  с помощью фабрик поставщиков 305
Встряхивание дерева
  стадия оптимизации 436
  целесообразность применения 436
Выражения в Angular,
    применение 68

Д

Декораторы
  @Component, применение 59
  @ContentChildren, применение 114
  @ContentChild, назначение
      и применение 109
  @Directive, применение 81
  @HostListener, применение 83
  @Injectable, применение 280
  @Input, применение 66
  @Output, применение 77
  @ViewChildren, применение 109, 217
  @ViewChild, назначение
      и применение 105
  компонентов Angular 2, порядок
      применения 59
  метаданные, описание 59
  определение 59
  применение 59
Директивы
  formControlName, применение 142
  formControl, назначение и
      применение 122
  formGroup, применение 141
  ng-content, назначение
      и применение 86
  ngController, преобразование 35
  ngForm, применение 136
  ngFor, назначение и применение 88
  ngIf, назначение и применение 88
  ngInclude, преобразование 34
  ngModelGroup, применение 139
  ngModel, назначение
      и применение 116, 137

routerLinkActiveOptions, применение 249
routerLinkActive, назначение и применение 248
routerLink, назначение и применение 238
routerOutlet, назначение и применение 236, 252
template, назначение и применение 90
взаимосвязь с данными, особенности 28
компонентов, реорганизация 33
ориентированная на контроллеры модель 28
присоединение
к элементам разметки 82
событий 83
разбиение на компоненты 29
составления форм, применение 144
структурные, декомпозиция 89

З

Загрузка по требованию
назначение 437
организация 439
Загрузчик модулей SystemJS, назначение и применение 45
Зоны
внедрение поведения 403
выполнение за пределами, организация 414
жизненный цикл, демонстрация 412
переопределяемые события, перечень 404
планирование, выполнение и отмена задач 405
формирование ядра, принцип 408

И

Интерфейсы
ActivatedRoute, назначение и применение 256
Angular CLI, назначение и применение 342
ArticleSourceInterface, назначение и применение 291
postMessage, назначение и применение 430
Routes
назначение и реализация 234
упорядочение маршрутов 242
Validator, назначение и реализация 149
перехвата из базового модуля, применение 99

К

Каналы чистые и нечистые, установка 398
Карты исходного кода
назначение 322
указание 323
формирование 323
Классы
AbstractControl, назначение и применение 147
BehaviorSubject, назначение и применение 202
ChangeDetectorRef, назначение и применение 220
ComponentFixture, назначение и применение 360
DelayValidator, назначение и применение 157
EventEmitter, назначение и применение 77
FormArray, назначение и применение 130
FormBuilder, назначение и применение 140
FormControl, назначение и применение 120, 223
FormGroup, назначение и применение 128
Http, назначение и методы 192
NgModule, назначение и применение 283
Observable, назначение и методы 191, 193, 198

OpaqueToken, назначение и применение 301
Promise
методы, применение 174, 177, 178
назначение и применение 161
QueryList, назначение и применение 218
Response, назначение и применение 191
Subject, назначение и методы 196
TestBed, применение и назначение 359
UpgradeModule, назначение и применение 45
ZoneSpec, назначение и применение 405
базовые для компонентов, определение 57
порядок определения 57
Компиляция
динамическая
на стадии разработки приложений 436
определение 432
исходного кода TypeScript, описание 321
перед выполнением
на стадии сборки и развертывания приложений 436
настройка компилятора ngc 433
определение 432
порядок перехода в приложении 435
применение 433
установка зависимостей 433
Компоненты
Angular 2
входные данные, объявление 66
как классы, определение 57
порядок определения 57
применение в приложениях Angular 1 48
самоописание, принцип 61
формы представлений 80
директивы, преобразование 36
моменты в жизненном цикле, перехват 408
назначение 28
основные, реализация в AngularJS 1.5 37
передача членов, процесс 66
перехватчики жизненного цикла
применение 99
разновидности 100
порожденные, ссылка на родительские 101
связывание, порядок 63

M

Маршрутизация
извлечение параметров маршрута 256
ограничители маршрутов, применение 268
определение маршрутов 234
основополагающие принципы в Angular 236
поведение сохраняющего состояние маршрута, реализация 248
порожденные и родительские маршруты, определение 253
предоставление маршрутов 235
установка базового URL 232
Матричные параметры URL
назначение 259
обработка 259
Микрозадачи и макрозадачи, назначение 409
Модели
'издатель-подписчик'
изящная реализация 215
составляющие и реализация 195
компонентов, идеальные 422
теневые по спецификации Shadow DOM, применение 423
Модули
Component Router, применение 232
FormsModule, применение 123
Http, применение 189
Observable, применение 188, 197
ReactiveFormsModule, применение 123, 223
RouterModule, применение 235
Validators, применение 123

Модульное тестирование
базовый принцип 365
без внедрения зависимостей, особенности 379
зависящих от служб компонентов
с помощью заглушек 383
средствами шпионов 387
служб, порядок 376

Н

Наблюдаемые объекты 181
в HTTP-запросах, применение 189
внедрение в Angular 2 188
в сравнении с обязательствами 189
'горячие' и 'холодные', определение 194
как поток событий 193
композиция, особенности 214
назначение 183
поддержка в библиотеке RxJS 181
преобразование в обязательства 183
сведение 227
Наблюдатели
назначение 188
определение 193
отличие от наблюдаемых объектов 197
Навигация
по ссылкам, реализация 239
с помощью службы Router, реализация 243

О

Обнаружение изменений
внесение корректив в схему 396
выполнение цикла
в зоне 419
в компоненте 419
принудительное, запрос 420
выполняемый цикл 399
применяемый процесс 396
Обязательства 160
барьеры, реализация 177
выполнение и отклонение, порядок 165
деревья функциональных возможностей, составление 172
закрытые члены 168
назначение 168, 183
нормализация 173
оболочки, создание 172
обработчики состояний
множественное определение 167
обмен данными 169
отклонение по цепочке 169
отложенное определение 167
связывание в цепочку 169
установка 163
отмены, внедрение 179
поддержка в браузерах 160
преобразование в наблюдаемые объекты 181
принцип выполнения, стадии 160
реализация в Angular 2, формы 161
смена состояний, порядок 166
совокупные, применение 176
Организация приложений Angular 2
для компиляции перед выполнением 434
конфигурирование TypeScript
карты исходного кода, формирование 322
компиляция, описание 321
свойства режима компиляции, описание 319
перенос приложения в сборку Webpack, порядок 334
способы и средства 312
формирование в интерфейсе Angular CLI 342

П

Пакеты
concurrently 356
Переменные шаблона
доступность 94
обращение к элементам разметки 91
Поставщики
маркеров, назначение 295
служб
директивные, назначение 298

порядок указания 293
список 283
схема в Angular 2 303
фабрики, назначение 305
Представления
виды инкапсуляции в Angular 426
порожденные
вложенные, определение 253
реализация 255
стандартные и профильные, применение 263
Привязка
данных
двунаправленная, реализация 116
к атрибуту, порядок 70
однонаправленная, реализация 68
формы 118
приемников к событиям, синтаксис 77
свойств атрибутов 95
событий
назначение 71
синтаксис 72
Проверка достоверности данных
асинхронное средство
особенности 156
создание 153
во всех полях формы, реализация 125
в одном поле формы, реализация 119
специальное средство, создание 146

P

Рабочие веб-процессы
абстракция в Angular 431
особенности 430
преимущество применения 431
средство визуализации, назначение 430

C

Сквозное тестирование
базовый принцип 366
построение объекта страницы как условие 370
средствами Protractor
особенности 367
установка и настройка среды тестирования 367
целевые компоненты, описание 373
Службы
Angular 2, внедрение в Angular 1 52
Router, назначение и применение 243
аутентификации, наблюдаемые
внедрение 202
построение 201
внедрение в компоненты
особенности 280
по псевдонимам 292
'издатель-подписчик', построение 209
имитация с помощью заглушек 303
назначение в Angular 43
нормализация типов 42
получение экземпляров, порядок 290
поставщики, определение 53
тестирование, порядок 374
типа NgZone
внедрение 411
назначение 410
типа Renderer, применение 428
События
естественные в браузерах
привязка 72
применение 76
самонарождающиеся, перехват 82
специальные
назначение 74
прием 77
распространение 76
Стилевое оформление
в Angular, возможности 422
естественная инкапсуляция 425
отмена инкапсуляции 425
эмуляция инкапсуляции 424
Стратегии обнаружения изменений
OnPush, определение 399, 419
определение 364
ресурсов
доступные типы 246
типа HashLocationStrategy, особенности 247

типа LocationStrategy, порядок выбора 245
типа PathLocationStrategy, особенности 247
Субъекты, определение 188

Т

Тестирование в Angular 2
инструментальные средства, описание 350
компиляция файлов и тестов в TypeScript 354
конфигурирование
Karma, порядок 352
PhantomJS, порядок 353
минимальный набор
модульных тестов, создание 351, 359
сквозных тестов, создание 371
написание тестового сценария 356
тестовая конфигурация, создание 361
Транспиляция в браузере
назначение 324
особенности 326
средствами SystemJS, порядок 325

У

Упаковщик Webpack
внедрение
в Karma, порядок 355
прокладок и полифилов 337
инструментальные средства, назначение 333
назначение и установка 334
подключаемый модуль html-webpack-plugin, применение 339
Устранение ложных повторений
входных данных, особенности 225
определение 225

Ф

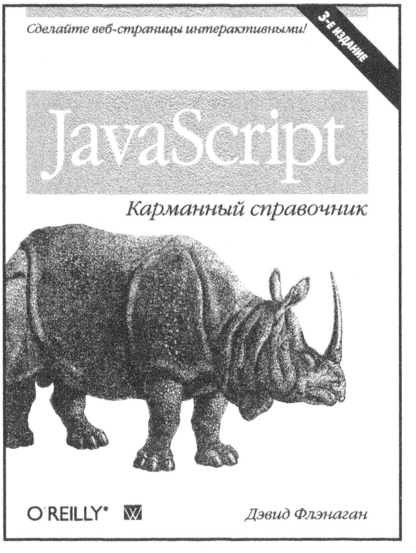
Фабрики поставщиков
назначение 306
определение 307
Файлы
инициализации рабочих веб-процессов, создание 428
компиляции TypeScript, единые, составление 323
конфигурации
package.json
порядок составления 312
разделы, описание 313
сценарии, описание 316
tsconfig.json, составление 318
объявлений TypeScript, назначение 318
приложений Angular 2
категории 346
описание 343
составление 327
Формы
назначение 115
оценка достоверности 125
сбор данных из полей, порядок 137
элементарные
разновидности 134
реализация 135, 143
Функции
async(), применение 360
beforeEach() и afterEach(), применение 352
catch() 172
describe(), применение 352
downgradeComponent(), применение 50
downgradeInjectable(), применение 54
enableProdMode(), применение 397
forwardRef(), применение 64, 106
inject(), применение 376
platformBrowser(), применение 433, 435
reject() 161
resolve() 161
исполнители обязательств, назначение 161

Ш

Шаблоны
'Наблюдатель'
назначение 188
описание 188
реализация в библиотеке RxJS 198

JAVASCRIPT
КАРМАННЫЙ СПРАВОЧНИК
3-Е ИЗДАНИЕ

Дэвид Флэнаган



www.williamspublishing.com

JavaScript — популярнейший язык программирования, который уже более 15 лет применяется для написания сценариев интерактивных веб-страниц. В книге представлены наиболее важные сведения о синтаксисе языка и показаны примеры его практического применения. Несмотря на малый объем карманного издания, в нем содержится все, что необходимо знать для разработки профессиональных веб-приложений.

Главы 1–9 посвящены описанию синтаксиса последней версии языка (спецификация ECMAScript 5).

- Типы данных, значения и переменные
- Инструкции, операторы и выражения
- Объекты и массивы
- Классы и функции
- Регулярные выражения

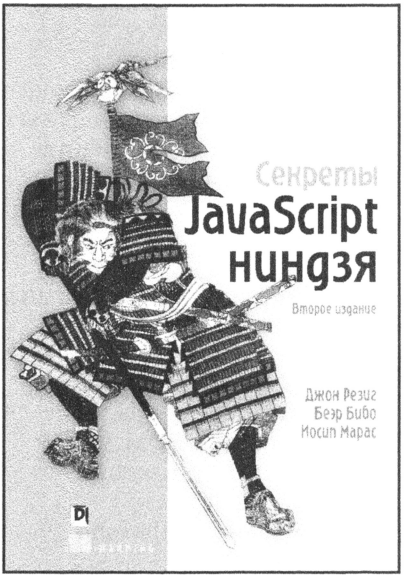
В главах 10–14 рассматриваются функциональные возможности языка наряду с моделью DOM и средствами поддержки HTML5.

ISBN 978-5-8459-1830-7

в продаже

СЕКРЕТЫ JAVASCRIPT НИНДЗЯ *2-е издание*

Джон Резиг
Беэр Бибо
Иосип Марас



www.dialektika.com

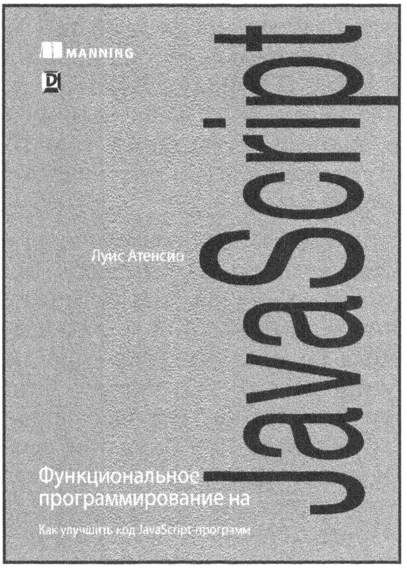
В этой книге раскрываются секреты мастерства разработки веб-приложений на JavaScript с учетом нововведений в стандартах ES6 и ES7. Начиная с пояснения таких основных понятий, как функции, объекты, замыкания, прототипы, регулярные выражения, обещания, события и таймеры, авторы постепенно проводят читателя по пути обучения от ученика до мастера, раскрывая немало секретов и специальных приемов программирования на конкретных примерах кода JavaScript. В книге уделяется немало внимания вопросам написания кросс-браузерного кода и преодолению связанных с этим типичных затруднений, что может принести немалую пользу всем, кто занимается разработкой веб-приложений.

Книга рассчитана на подготовленных читателей, стремящихся повысить свой уровень мастерства в программировании на JavaScript в частности и разработке веб-приложений вообще.

ISBN 978-5-9908911-8-0 **в продаже**

ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ НА JAVASCRIPT

Луис Атенсио



www.williamspublishing.com

В этой книге рассматриваются решения самых разных задач средствами функционального программирования на JavaScript, включая обработку коллекций, AJAX-запросов, обращений к базе данных, событий и пр. На конкретных примерах в ней поясняются основные понятия ФП, в том числе декомпозиция и композиция, чистые функции, связывание функций в цепочку, конвейеры функций, карринг, функторы, монады, отложенное вычисление и прочие. Усвоив теоретические основы функционального программирования, а также примеры программ, демонстрирующие практическое применение методик ФП, читатель научится мыслить функционально и, приобретя некоторую практику, станет делать это интуитивно.

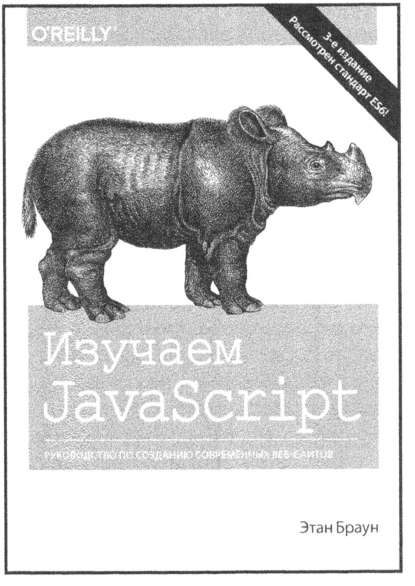
Книга адресована разработчикам, твердо усвоившим основы программирования на JavaScript и обладающим достаточным опытом проектирования веб-приложений.

ISBN 978-5-9909445-8-9 **в продаже**

ИЗУЧАЕМ JAVASCRIPT. РУКОВОДСТВО ПО СОЗДАНИЮ СОВРЕМЕННЫХ ВЕБ-САЙТОВ

3-е издание

Этан Браун



www.dialektika.com

Эта книга — хорошо написанное сжатое введение в язык *JavaScript*, включая стандарт *ECMAScript 6*. Она знакомит программистов (любителей и профессионалов) со спецификацией *ES6* наряду с некоторыми связанными с ней инструментальными средствами и методиками на сугубо практической основе.

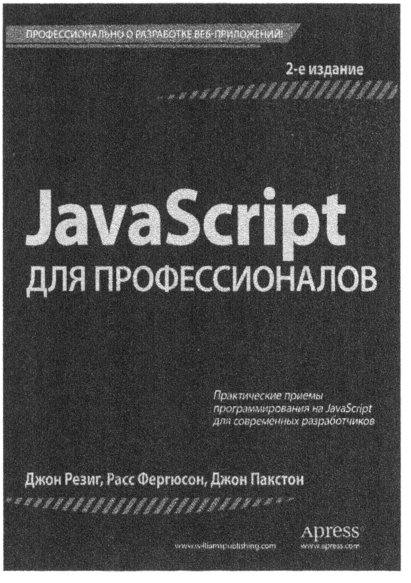
Эта книга предназначена, прежде всего, для читателей, уже обладающих некоторым опытом программирования (освоивших хотя бы вводный курс программирования). Опытные программисты найдут здесь практически полное описание важнейших концепций *JavaScript*, введенных в стандарт *ES6*. Программистам, переходящим на *JavaScript* с другого языка, содержимое этой книги также должно понравиться.

ISBN 978-5-9908463-9-5 **в продаже**

JavaScript для профессионалов

Второе издание

Джон Резиг,
Расс Фергюсон,
Джон Пакстон



www.williamspublishing.com

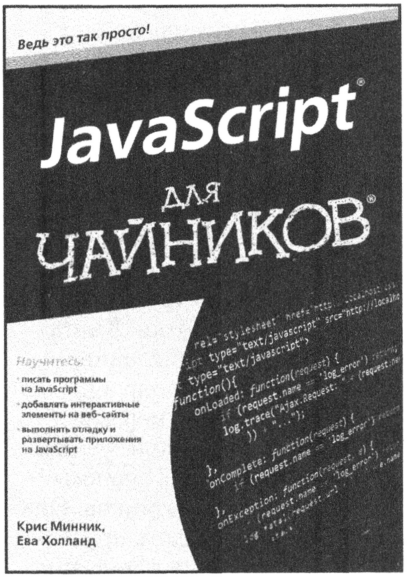
Эта книга является незаменимым пособием для профессиональных разработчиков современных веб-приложений на JavaScript. Читатель найдет в ней все, что требуется знать о современном состоянии языка JavaScript, его достоинствах и недостатках, новых языковых средствах, внедренных в последних версиях стандарта ECMAScript, передовых приемах отладки и тестирования кода, а также инструментальных средствах разработки. Книга изобилует многочисленными практическими и подробно разбираемыми примерами кода, повторно используемых функций и классов, экономящих время разработчиков. Она помогает им овладеть практическими навыками написания динамических веб-приложений на высоком профессиональном уровне, а также повысить свою квалификацию.

Книга рассчитана на тех, кто интересуется разработкой веб-приложений и имеет опыт программирования на JavaScript.

ISBN 978-5-8459-2054-6 **в продаже**

JAVASCRIPT ДЛЯ ЧАЙНИКОВ

Крис Минник,
Ева Холланд



www.dialektika.com

JavaScript — ключевой инструмент создания современных сайтов, и благодаря данному руководству, ориентированному на новичков, вы сможете изучить язык в короткие сроки и с минимумом усилий. Узнайте, какова структура языка, как правильно записывать его инструкции, как применять CSS, работать с онлайн-графикой и подключать программные интерфейсы HTML5. Все темы можно закрепить практическими упражнениями, доступными для выполнения на сайте Codecademy.com.

Основные темы книги:

- как настроить среду разработки;
- для чего нужны массивы;
- применение циклов;
- использование библиотеки jQuery;
- создание анимации в JavaScript;
- работа с CSS и графикой;
- AJAX и JSON;
- как избежать распространенных ошибок.

ISBN 978-5-8459-2036-2 **в продаже**

JAVASCRIPT ДЛЯ ЛЮБОЗНАТЕЛЬНЫХ

Крис Минник,
Ева Холланд

www.dialektika.com

Язык программирования JavaScript оживляет веб-страницы и позволяет запускать на них интерактивные приложения, эффективно взаимодействующие с пользователями. В этой книге вы узнаете, как с помощью JavaScript написать рассказ, разработать веб-страницу, создать игру в слова, построить калькулятор, организовать лавку по продаже лимонада и выполнить другие не менее захватывающие проекты. Удивите своих друзей!

- Используйте консоль JavaScript своего браузера в самых первых проектах.
- Применяйте полученные навыки. Создавайте полезные приложения и увлекательные игры, чтобы произвести впечатление на своих друзей.
- Не останавливайтесь на достигнутом. Проявите творческий подход к написанию приложений и разработке веб-сайтов.

ISBN 978-5-9908463-5-7 **в продаже**

СОВРЕМЕННЫЙ ВЕБ-ДИЗАЙН НАСТОЛЬНЫЙ И МОБИЛЬНЫЙ
3-Е ИЗДАНИЕ

Ю.А. Сырых

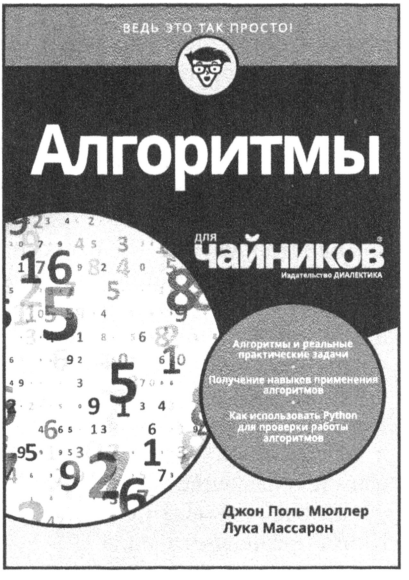
www.dialektika.com

Эта книга предназначена для начинающих веб-дизайнеров. Она описывает основные правила и тонкости дизайнерской работы на всех этапах разработки сайта — от постановки задачи, отбора материала и разработки макета, до тестирования готового сайта и публикации его в сети. Читатель найдет здесь знания и рекомендации, критически важные для успешного и грамотного дизайнера, подробное описание и примеры основных современных стилей веб-сайтов, множество небольших, но очень полезных советов. В целом книга поможет начинающему специалисту не только успешно нарисовать неплохой коммерческий дизайн, но и прийти к пониманию сути веб-дизайна, усвоить основные принципы разработки дизайн-макетов сайтов в различных стилях. Все это позволит ему успешно двигаться дальше, творчески расти и в дальнейшем разрабатывать собственные стили оформления сайтов.

ISBN 978-5-8459-1905-2 **в продаже**

АЛГОРИТМЫ ДЛЯ ЧАЙНИКОВ

Джон Поль Мюллер
Лука Массарон



www.dialektika.com

Эта книга — действительно книга для чайников, поскольку основная ее задача не научить программировать реализации тех или иных давно известных алгоритмов, а познакомить вас с тем, что же такое алгоритмы, как они влияют на нашу повседневную жизнь, и каково состояние дел в этой области человеческих знаний сегодня.
В книге рассматривается крайне широкий спектр вопросов, связанных с алгоритмами — это и стандартные сортировка и поиск, и работа с графами (но с уклоном не в стандартные базовые алгоритмы, а в приложении их к таким явлениям сегодняшнего дня, как, например, социальные сети), работа с большими данными и вопросы искусственного интеллекта.
При этом материал книги — не просто отвлеченный рассказ о том или ином аспекте современных алгоритмов, но и демонстрация реализаций алгоритмов с конкретными примерами на языке программирования Python.
Книга будет полезна всем, кто интересуется современным состоянием дел в области программирования и алгоритмов.

ISBN 978-5-9909446-2-6

в продаже

ПРАКТИКА СИСТЕМНОГО И СЕТЕВОГО АДМИНИСТРИРОВАНИЯ
ТОМ 1
ТРЕТЬЕ ИЗДАНИЕ

Томас А. Лимончелли
Кристина Дж. Хоган
Страта Р. Чейлап

www.williamspublishing.com

В книге описана практика авторитетных системных администраторов, имеющих многолетний опыт работы. В ней изложены стратегические решения и концепции, поучительные истории и рекомендации, касающиеся системного и сетевого администрирования, в частности, вопросы автоматизации и централизации процессов инсталляции и поддержки, управления парком рабочих станций, серверами и службами, а также проблемы хранения данных и менеджмента (этические нормы, наем и увольнение сотрудников, управление временем и т.д.). Книга предназначена для системных администраторов всех уровней, которые стремятся освоить современные методы и стратегии. Она также будет полезной для менеджеров, управляющих работой системных администраторов.

ISBN 978-5-6040043-1-9 **в продаже**

ПРИКЛАДНОЕ МАШИННОЕ ОБУЧЕНИЕ С ПОМОЩЬЮ SCIKIT-LEARN И TENSORFLOW

Орельен Жерон

www.dialektika.com

Благодаря серии недавних достижений глубокое обучение значительно усилило всю область машинного обучения. В наше время даже программисты, почти ничего не знающие об этой технологии, могут использовать простые и эффективные инструменты для реализации программ, которые способны обучаться на основе данных. В настоящем практическом руководстве показано, что и как следует делать. За счет применения конкретных примеров, минимума теории и двух фреймворков Python производственного уровня — Scikit-Learn и TensorFlow — автор книги Орельен Жерон поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем. Вы узнаете о ряде приемов, начав с простой линейной регрессии и постепенно добравшись до глубоких нейронных сетей. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования.

ISBN 978-5-9500296-2-2

в продаже

JAVA SE 9
БАЗОВЫЙ КУРС
ВТОРОЕ ИЗДАНИЕ

Кей С. Хорстманн

В этом кратком руководстве рассматриваются основные языковые средства и нововведения в версии Java SE 9, главным из которых является новая модульная система на платформе Java. На практических примерах исходного кода поясняются основные языковые средства, новшества, усовершенствования и прочие незначительные изменения в версии Java SE 9. Книга рассчитана как на начинающих, так и на опытных программистов, стремящихся писать в недалекой перспективе надежный, эффективный и безопасный код на Java.

www.williamspublishing.com

ISBN 978-5-6040043-0-2 **в продаже**

Angular. Сборник рецептов

В каркасе Angular 2 введена совершенно новая парадигма разработки веб-приложений. В нем полностью охвачены новейшие понятия, на которых будут создаваться браузеры следующего поколения, а также исключено все лишнее из версии Angular 1. Эта книга погружает читателя в саму суть наиболее важных понятий Angular 2. Помимо самого каркаса Angular 2, в книге рассматриваются новый синтаксис стандарта ES6 языка JavaScript, особенности языка TypeScript, спецификации Web Components, наблюдаемых объектов из библиотеки RxJS и много других новых понятий.

Освоить каркас Angular 2, без сомнения, нелегко, особенно тем, кто лишь начинает открывать для себя новые технологии и стандарты. В этой книге рассматриваются самые сложные понятия Angular 2 и в то же время демонстрируются самые передовые нормы практики, помогающие овладеть эффективными инструментальными средствами данного каркаса. Кроме того, в ней поясняются методы и принципы, ускоряющие написание веб-приложений, а также часто пренебрегаемые вопросы тестирования и производительности. Эта книга позволит читателям досконально усвоить особенности каркаса Angular 2.

Основные темы книги

- Как лучше всего перенести приложение Angular 1 в Angular 2
- Описание базовых элементов Angular 2, таких как компоненты, формы и службы
- Овладение такими сложными вещами, как наблюдаемые объекты и обязательства
- Корректная реализация веб-приложений с применением таких передовых методик, как внедрение зависимостей
- Как повысить до максимума производительность приложений Angular 2
- Освоение наилучших способов перехода от исходного кода TypeScript приложения Angular 2 к его версии, полноценно функционирующей на веб-сайте
- Передовые нормы практики оптимизации и тестирования крупных приложений Angular 2

Категория: программирование
Предмет рассмотрения: разработка веб-приложений в Angular 2
Уровень: промежуточный–продвинутый

ДИАЛЕКТИКА
http://www.williamspublishing.com

Packt
Packt Publishing Ltd.
www.packtpub.com

ISBN 978-5-9909446-6-4