Изучаем TypeScript 3

TypeScript – это и язык, и набор инструментов для генерации кода JavaScript. Он был разработан Андерсом Хейлсбергом в корпорации Microsoft, чтобы помочь разработчикам в написании кода JavaScript в масштабах предприятия.

Книга начинается со знакомства с языком TypeScript, и, поэтапно переходит от базовых понятий к продвинутым и мощным функциям языка, включая методы асинхронного программирования, декораторы и обобщения. Также параллельно рассматривается множество современных фреймворков JavaScript и TypeScript - для каждого из них подробно описано модульное и интеграционное тестирование. Описаны некоторые из известных объектно-ориентированных методов и шаблонов проектирования, а также представлены их реальные реализации.

К концу книги вы создадите всеобъемлющее комплексное веб-приложение, которое покажет, как можно объединить в реальном сценарии возможности языка TypeScript, шаблоны проектирования и передовые практики разработки.

С помощью этой книги вы:

- изучите как основы, так и продвинутые возможности языка TypeScript;
- осуществите интеграции существующих библиотек JavaScript и сторонних фреймворков с использованием файлов объявлений;
- освоите такие популярные JavaScript-фреймворки, как Angular, React и др.;
- создадите наборы тестов для своего приложения с помощью Jasmine и Selenium;
- организуете код своего приложения, используя модули, загрузчики AMD и SystemJS;
- изучите передовые принципы объектно-ориентированного проектирования;
- сравните различные реализации концепции MVC в Aurelia, Angular, React и др.

ISBN 978-5-97060-757-2

9 785970 607572 >

Изучаем TypeScript 3

Натан Розенталс

Изучаем TypeScript 3

ДМК ИЗДАТЕЛЬСТВО

Натан Розенталс

Изучаем TypeScript 3

Создавайте промышленные веб-приложения корпоративного класса с использованием TypeScript 3 и современных фреймворков

Mastering TypeScript 3

Build enterprise-ready, industrial-strength web applications using TypeScript 3 and modern frameworks

Nathan Rozentals

Third Edition

BIRMINGHAM • MUMBAI

Packt>

Изучаем TypeScript 3

Создавайте промышленные веб-приложения корпоративного класса с использованием TypeScript 3 и современных фреймворков

Натан Розенталс

Москва, 2019 ДМК Пресс издательство

УДК 004.43
ББК 32.973.3
Р64

Р64 Натан Розенталс
Изучаем TypeScript 3. – М.: ДМК Пресс, 2019. – 624 с.

ISBN 978-5-97060-757-2

TypeScript – это и язык, и набор инструментов для генерации кода JavaScript. Язык TypeScript и его компилятор завоевали прочные позиции в сообществе разработчиков на JavaScript и продолжают набирать силу, имея богатый инструментарий разработки. Многие масштабные проекты на JavaScript, в том числе проекты Adobe, Mozilla и Asana, приняли решение перевести свою кодовую базу с JavaScript на TypeScript.

Эта книга представляет собой руководство по TypeScript, которое начинается с базовых понятий, а затем представляет более продвинутые возможности языка. Подробно рассказано об использовании TypeScript со множеством современных фреймворков, применяются методы разработки через тестирование, дано много стандартных шаблонов проектирования. Итогом изучения будет полностью готовое к использованию приложение на TypeScript.

Издание будет полезно всем разработчикам приложений.

УДК 004.43
ББК 32.973.3

ISBN 978-5-97060-757-2 (рус.)
ISBN 978-1-78953-670-6 (анг.)

Об авторе

Натан Розенталс (**Nathan Rozentals**) занимается созданием коммерческого программного обеспечения на протяжении более 30 лет. Как и многие программисты того времени, он помог спасти мир в 2000 году.

Работая, он пытался освоить множество объектно-ориентированных языков, начиная с простого старого C, переходя на C++, Java, а затем C#. В TypeScript он нашел язык, в который можно привнести все шаблоны объектно-ориентированного проектирования и принципы, которые он изучил за годы работы с JavaScript.

Когда он не программирует, он думает о программировании. Чтобы перестать думать о нем, он занимается виндсерфингом, играет в футбол или просто смотрит, как в футбол играют профессионалы. Они делают это намного лучше, чем он.

Я хотел бы поблагодарить своего партнера Кэти за то, что была рядом со мной, и в горе, и в радости, что научила меня, что значит любовь. Эйрон и Дейна, я думаю о вас, ребята, каждый день, и так горжусь каждым из вас. Мэтт, просто продолжай быть собой.

Всем в Vix спасибо за то, что сделали работу такой приятной. То, что мы делаем, нелегко, но это стоит того. Я никогда не встречал такого количества талантливых людей, сосредоточенных в одном месте.

О рецензентах

Гаурав Аророаа (Gaurav Aroraa) имеет степень магистра компьютерных наук и статус MVP, является пожизненным членом Компьютерного общества Индии (CSI), консультативным членом IndiaMentor, имеет сертификаты скрам-тренера, XEN-библиотеки – фонда инфраструктуры информационных технологий (ITIL-F) и сертификаты PRINCE-F и PRINCE-P, заверенные APMG Int. Гаурав – разработчик приложений с открытым исходным кодом и основатель Ovatic Systems. Недавно Гаурав был удостоен награды *Иконы года – за выдающиеся достижения в области стартапов в сфере технологии наставничества* за 2018–2019 гг. от Radio City – по инициативе Jagran за выдающуюся работу в течение 20-летней карьеры в отрасли в области технологического наставничества. Вы можете написать Гаураву в Twitter (`@g_arora`).

Ручитха Сахабанду (**Ruchitha Sahabandu**) – инженер-программист, технический инструктор и наставник с более чем 14-летним опытом проектирования и разработки бизнес-приложений и встроенных систем, основанных на платформах JavaScript, JEE, Oracle и технологиях C/C++. Он работал над многими сложными проектами, функционирующими на Agile и TDD, методологиями SDLC в таких областях, как автоматизированная система сбора платы за проезд, планирование ресурсов, управление цепями поставок и автоматизация производственных площадей для клиентов по всему миру.

В 2006 году он окончил Университет Коломбо, Шри-Ланка, по специальности «Компьютерные науки». В настоящее время он живет в Перте, Австралия, и трудится вместе с Натаном Розенталсом над разработкой продукта AFC следующего поколения.

Оглавление

Предисловие 19

Глава 1 Инструменты TypeScript и параметры фреймворков 25

Что такое TypeScript? 27
JavaScript и ECMAScript 28
Преимущества TypeScript 29
Компиляция 29
Сильная типизация 30
Синтаксический сахар 31
Определение типов для популярных библиотек JavaScript 32
DefinitelyTyped 33
Инкапсуляция 33
Классы TypeScript генерируют замыкания 35
Методы доступа public и private 35

Интегрированные среды разработки TypeScript 37
Компиляция на основе Node 38
Создание файла tsconfig.json 39
Локализованные сообщения 40
Visual Studio Code 41
Установка VSCode 41
Изучение VSCode 41
Создание файла tasks.json 41
Сборка проекта 44
Создание файла launch.json 44
Установка точек останова 44
Отладка веб-страниц 45
Microsoft Visual Studio 48
Создание проекта в Visual Studio 48
Настройки проекта по умолчанию 50
Отладка в Visual Studio 53
WebStorm 54
Создание проекта в WebStorm 55
Файлы по умолчанию 55
Создание простого HTML-приложения 56
Запуск веб-страницы в Chrome 57
Другие редакторы 59
Использование --watch и Grunt 59

Резюме 62

Глава 2 Типы, переменные и методы функций 63

Базовые типы 64
Типизация в JavaScript 64
Типизация в TypeScript 65
Синтаксис типов 66
Типизация с поддержкой вывода типов 69
Утиная типизация 70

Шаблонные строки 71
Массивы 72
for...in и for...of 73
Тип any 74
Явное приведение типов 75
Перечисления 76
Const enum 77
Строковые перечисления 78
Реализация перечислений 78
Const 80
Ключевое слово let 80
Определенное присваивание 83
Типы свойств с точечной нотацией 84
Числовые разделители 85
Функции 85
Типы возвращаемого значения 85
Анонимные функции 86
Необязательные параметры 87
Параметры по умолчанию 88
Оставшиеся параметры 89
Функции обратного вызова 91
Сигнатуры функций 93
Переопределение функций 95
Try...catch 96
Расширенные типы 97
Объединенные типы 98
Охранники типов 98
Псевдонимы типов 100
Null и undefined 101
Нулевые операнды 103
Never 104
Unknown 105
Object rest and spread 106
Приоритет распространения 107
Использование операторов остатка и распространения с массивами 108
Кортежи 109
Деконструкция кортежей 110
Необязательные элементы кортежа 111
Кортежи и синтаксис оператора остатка 111
Bigint 112
Резюме 116
Глава 3 Интерфейсы, классы и наследование 117
Интерфейсы 118
Необязательные свойства 119
Компиляция интерфейса 120

Слабые типы 120
Вывод типов с помощью оператора in 121
Классы 122
Свойства класса 124
Реализация интерфейсов 124
Конструкторы классов 126
Функции класса 127
Определения функций интерфейса 130
Модификаторы класса 131
Модификаторы доступа конструктора 132
Свойство readonly 134
Методы доступа к свойствам класса 134
Статические функции 136
Статические свойства 136
Пространства имен 137
Наследование 138
Наследование интерфейса 139
Наследование классов 139
Ключевое слово super 140
Переопределение функции 141
Члены класса protected 142
Абстрактные классы 143
Замыкания JavaScript 146
instanceof 148
Использование интерфейсов, классов и наследования – шаблон проектирования Factory 150
Бизнес-требования 150
Что делает шаблон проектирования Factory 150
Интерфейс IPerson 151
Класс Person 151
Классы специалистов 152
Класс Factory 153
Использование класса Factory 154
Резюме 155
Глава 4 Декораторы, обобщения и асинхронные функции 156
Декораторы 157
Синтаксис декораторов 158
Несколько декораторов 159
Фабрика декораторов 159
Параметры декораторов класса 160
Декораторы свойств 162
Декораторы статических свойств 163
Декораторы методов 164
Использование декораторов методов 165
Декораторы параметров 167

Метаданные декораторов 168
Использование метаданных декораторов 170
Обобщения 171
Синтаксис обобщений 172
Создание экземпляра обобщенного класса 172
Использование типа T 174
Ограничение типа T 176
Обобщенные интерфейсы 178
Создание новых объектов в обобщениях 179
Расширенные типы с обобщениями 181
Условные типы 182
Распределенные условные типы 185
Выведение условных типов 187
keyof 189
keyof с числом 190
Отображаемые типы 192
Partial, Readonly, Record и Pick 193
Асинхронное программирование 195
Промисы 195
Синтаксис промисов 197
Использование промисов 199
Механизм обратного вызова в сравнении с синтаксисом промиса 200
Возвращение значений из промисов 201
async и await 203
awaitError 204
Синтаксис промиса в сравнении с синтаксисом async await 205
awaitMessage 206
Резюме 207
Глава 5 Файлы объявлений и строгие опции компилятора 208
Глобальные переменные 209
Использование блоков кода JavaScript в HTML 211
Структурированные данные 212
Пишем свой файл объявлений 214
Ключевое слово module 216
Интерфейсы 218
Объединенные типы 220
Слияние модулей 221
Справочник синтаксиса объявлений 222
Переопределение функций 222
Синтаксис JavaScript 222
Синтаксис файла объявлений 223
Вложенные пространства имен 223
Синтаксис JavaScript 223
Синтаксис файла объявлений 223

Классы 223
Синтаксис JavaScript 223
Синтаксис файла объявлений 223
Пространства имен классов 224
Синтаксис JavaScript 224
Синтаксис файла объявлений 224
Перегрузки конструктора класса 224
Синтаксис JavaScript 224
Синтаксис файла объявлений 224
Свойства класса 224
Синтаксис JavaScript 225
Синтаксис файла объявлений 225
Функции класса 225
Синтаксис JavaScript 225
Синтаксис файла объявлений 225
Статические свойства и функции 225
Синтаксис JavaScript 225
Синтаксис файла объявлений 226
Глобальные функции 226
Синтаксис JavaScript 226
Синтаксис файла объявлений 226
Сигнатуры функций 226
Синтаксис JavaScript 226
Синтаксис файла объявлений 226
Необязательные свойства 226
Синтаксис JavaScript 227
Синтаксис файла объявлений 227
Слияние функций и модулей 227
Синтаксис JavaScript 227
Синтаксис файла объявлений 227
Строгие опции компилятора 227
noImplicitAny 228
strictNullChecks 229
strictPropertyInitialization 230
noUnusedLocals и noUnusedParameters 231
noImplicitReturns 232
noFallthroughCasesInSwitch 233
strictBindCallApply 234
Резюме 236
Глава 6 Сторонние библиотеки 237
Использование файлов определений 237
Использование NuGet 238
Использование диспетчера расширений NuGet 238
Установка файлов объявлений 240
Использование консоли диспетчера пакетов 241

Установка пакетов 241
Поиск имен пакетов 241
Установка конкретной версии 242
Использование npm и @types 242
Использование сторонних библиотек 243
Выбор фреймворка JavaScript 244
Backbone 245
Использование наследования с Backbone 245
Использование интерфейсов 248
Использование синтаксиса обобщений 248
Использование ECMAScript 5 249
Совместимость Backbone с TypeScript 250
Angular 1 250
Классы Angular и $scope 252
Совместимость Angular 1 с TypeScript 254
Наследование – Angular 1 против Backbone 254
ExtJS 255
Создание классов в ExtJS 256
Использование приведения типов 257
Компилятор TypeScript специально для ExtJS 258
Резюме 259
Глава 7 Фреймворки, совместимые с TypeScript 260
Что такое MVC? 261
Модель 261
Представление 262
Контроллер 263
Резюмируя 264
Преимущества использования MVC 265
Пример приложения 265
Использование Backbone 267
Производительность визуализации 267
Настройка Backbone 269
Структура Backbone 269
Модели Backbone 270
Класс ItemView 273
Класс ItemCollectionView 275
Приложение Backbone 277
Формы 278
Резюмируя 282
Использование Aurelia 282
Настройка Aurelia 282
Контроллеры и модели Aurelia 283
Представления Aurelia 284
Начальная загрузка приложения 285

События 286
Формы 287
Резюмируя 288
Angular 288
Установка Angular 288
Модели Angular 289
Представления Angular 290
События 292
Формы 293
Формы шаблонов 293
Ограничения форм шаблонов 295
Реактивные формы 295
Использование реактивных форм 296
Резюмируя 298
Использование React 299
Настройка React 299
Настройка webpack 301
ItemView 304
CollectionView 306
Начальная загрузка 309
Формы 311
Изменение состояния 312
Свойства состояния 313
Возможности TypeScript в React 316
Синтаксис оставшихся параметров 316
Свойства по умолчанию 317
Резюмируя 318
Сравнение производительности 318
Резюме 321
Глава 8 Разработка через тестирование 322
Разработка через тестирование 323
Модульные, интеграционные и приемочные тесты 325
Модульные тесты 325
Интеграционные тесты 325
Приемочные тесты 326
Фреймворки для модульного тестирования 326
Jasmine 327
Простой тест 328
Репортеры 331
Сопоставители 332
Запуск и завершение теста 334
Принудительные тесты 335
Пропуск тестов 336
Тесты, управляемые данными 337
Использование шпионов 340

Слежка за функциями обратного вызова 341
Использование шпионов в качестве фальшивок 342
Асинхронное тестирование 343
Использование функции done() 345
Использование async awaitc 347
HTML-тесты 348
Фикстуры 350
События DOM 351
Библиотеки для модульного тестирования 352
Testem 353
Karma 354
Тестирование в режиме headless 356
Protractor 357
Selenium 357
Поиск элементов страницы 359
Использование непрерывной интеграции 361
Преимущества непрерывной интеграции 361
Выбор сервера сборки 362
Team Foundation Server 363
Jenkins 363
TeamCity 363
Отчеты об интеграционных тестах 363
Резюме 365
Глава 9 Тестирование фреймворков, совместимых с Typescript 366
Тестирование нашего приложения 366
Тестирование Backbone 367
Настройка теста 367
Тесты моделей 368
Тесты сложных моделей 371
Тесты визуализации 372
Тесты событий DOM 374
Тесты представления коллекции 375
Тесты формы 377
Резюмируя 379
Тестирование Aurelia 379
Настройка 379
Модульные тесты 380
Тесты визуализации 381
События DOM 385
Резюмируя 386
Тестирование Angular 386
Настройка 386
Тесты моделей 389
Тесты визуализации 390
Тесты форм 393

Резюмируя 395
Тестирование с React 396
Несколько точек входа 396
Использование Jest 397
Тесты начального состояния 400
Элемент input 402
Отправка формы 403
Подводя итоги 405
Резюме 405
Глава 10 Модуляризация 407
Основы 408
Экспорт модулей 410
Импорт модулей 411
Переименование модулей 411
Экспорт по умолчанию 412
Экспорт переменных 413
Типы импорта 414
Асинхронное определение модуля 415
Компиляция 416
Установка модуля AMD 417
Настройка Require 418
Настройка браузера 419
Зависимости модуля AMD 420
Начальная загрузка Require 423
Исправление ошибок конфигурации 424
Неправильные зависимости 425
Ошибки 404 425
Загрузка модулей с помощью SystemJS 426
Установка SystemJS 426
Конфигурация браузера 426
Зависимости модулей 429
Начальная загрузка Jasmine 431
Использование Express с Node 432
Установка Express 432
Использование модулей с Express 434
Маршрутизация 435
Шаблонизаторы 437
Использование Handlebars 438
События POST 441
Перенаправление HTTP-запросов 445
Функции Lambda 447
Архитектура функции Lambda 447
Настройка AWS 449
Serverless 451
Настройка Serverless 452

Развертывание 453
Функции Lambda в TypeScript 456
Node-модули функции Lambda 458
Логирование 460
Тестирование REST 461
Резюме 464
Глава 11 Объектно-ориентированное программирование 465
Принципы объектно-ориентированного программирования 466
Программирование в соответствии с интерфейсом 466
Принципы SOLID 467
Единственная ответственность 467
Открытость/закрытость 467
Принцип подстановки Барбары Лисков 468
Разделение интерфейса 468
Инверсия зависимостей 468
Проектирование пользовательского интерфейса 468
Концептуальный дизайн 469
Настройка Angular 471
Использование Bootstrap и Font-Awesome 472
Создание боковой панели 473
Создание наложения 477
Координация переходов 479
Шаблон State 480
Интерфейс шаблона State 481
Конкретные состояния 482
Шаблон Mediator 483
Модульный код 484
Компонент Navbar 484
Компонент SideNav 486
Компонент RightScreen 487
Дочерние компоненты 490
Реализация интерфейса посредника 490
Класс Mediator 491
Использование Mediator 495
Реагирование на события DOM 496
Резюме 498
Глава 12 Внедрение зависимости 499
Отправка почты 500
Использование nodemailer 501
Использование локального SMTP-сервера 502
Служебный класс 502
Настройки конфигурации 505
Зависимость объектов 507
Service Location 507

Антишаблон Service Location 510
Внедрение зависимости 510
Создание инжектора зависимостей 511
Разрешение интерфейса 511
Разрешение enum 511
Разрешение класса 512
Внедрение конструктора 515
Внедрение декораторов 516
Использование определения класса 517
Синтаксический анализ параметров конструктора 518
Поиск типов параметров 520
Внедрение свойств 520
Использование внедрения зависимости 521
Рекурсивное внедрение 522
Резюме 524
Глава 13 Создание приложений 525
Интеграция Node и Angular 526
Сервер Express 528
Конфигурация сервера 530
Ведение журнала сервера 531
Опыт взаимодействия 535
Использование Brackets 536
Использование Emmet 537
Создание панели входа 540
Аутентификация 542
Маршрутизация в Angular 543
Использование HTML-кода, предназначенного для пользовательского интерфейса 546
Стражи аутентификации 547
Связывание формы входа 549
Использование HttpClient 551
Использование Observable 553
Использование JWT-токенов 557
Верификация токенов 560
Использование Observables в гневе – of, pipe и map 561
Внешняя аутентификация 565
Получение API-ключа Google 565
Настройка социального логина 567
Использование данных пользователя Google 568
Резюме 570
Глава 14 Переходим к практике 572
Приложение Board Sales 573
API на основе базы данных 574
Структура базы данных 575

Конечные точки API 577
Параметризованные конечные точки API 581
Службы REST в Angular 584
Спецификация OpenAPI 587
Приложение BoardSales 588
Компонент BoardList 588
Визуализация данных REST 591
concatMap 594
forkJoin 599
Модульное тестирование Observables 602
Шаблон проектирования Domain Events 605
Вызов и использование событий предметной области 608
Фильтрация данных 612
Резюме 620
Указатель 621

Предисловие

Язык TypeScript и его компилятор имели огромный успех с момента своего выхода в конце 2012 года. Они быстро завоевали прочные позиции в сообществе разработчиков на JavaScript и продолжают набирать силу. Многие масштабные проекты на JavaScript, в том числе проекты Adobe, Mozilla и Asana, приняли решение перевести свою кодовую базу с JavaScript на TypeScript. В 2014 году команды Microsoft и Google объявили, что Angular 2.0 будет разрабатываться с использованием TypeScript, тем самым объединяя языки AtScript от Google и TypeScript от Microsoft в один.

Такое крупномасштабное отраслевое внедрение TypeScript показывает ценность данного языка, гибкость компилятора и повышение производительности, которое может быть достигнуто с помощью его богатого набора инструментов разработки. Помимо отраслевой поддержки, стандарты ECMAScript 6 и ECMAScript 7 становятся все ближе и ближе к публикации, а TypeScript предоставляет способ использования функций этих стандартов в наших приложениях сегодня.

Написание приложений на TypeScript стало еще более привлекательным благодаря большой коллекции файлов объявлений, созданных сообществом TypeScript. Эти файлы беспрепятственно интегрируют широкий спектр существующих фреймворков JavaScript в среду разработки TypeScript, что повышает производительность, принося с собой заблаговременное обнаружение ошибок и расширенные функции IntelliSense.

Однако язык JavaScript не ограничивается веб-браузерами. Теперь мы можем писать на JavaScript на стороне сервера, управлять приложениями для мобильных телефонов с помощью JavaScript и даже управлять микроустройствами, разработанными для интернета вещей, используя JavaScript. Поэтому все эти цели JavaScript достижимы для разработчика, пишущего на TypeScript, потому что TypeScript генерирует JavaScript.

Для кого эта книга

Эта книга представляет собой руководство по языку TypeScript, которое начинается с базовых понятий, а затем основывается на этих знаниях, чтобы представить более продвинутые возможности языка и фреймворки. Предварительных знаний JavaScript не требуется, хотя предполагается, что читатель обладает предварительными навыками программирования. Если вы хотите изучать TypeScript, эта книга предоставит вам все необходимые знания и навыки для работы с любым проектом на TypeScript. Если вы уже являетесь опытным разработчиком на JavaScript или TypeScript, то эта книга выведет ваши навыки на новый уровень. Вы узнаете, как использовать TypeScript со множеством современных фрейм-

ворков, и выберете лучший фреймворк, соответствующий требованиям вашего проекта. Вы будете исследовать методы разработки через тестирование, изучите стандартные шаблоны проектирования и узнаете, как собрать полностью готовое к использованию приложение на TypeScript.

О чем рассказывается в этой книге

`Глава 1` «*Инструменты TypeScript и параметры фреймворков*» подготавливает почву для начала разработки на TypeScript. В ней рассматриваются преимущества использования TypeScript в качестве языка и компилятора, а затем рассматривается создание полной среды разработки с использованием ряда популярных интегрированных сред разработки.

`Глава 2` «*Типы, переменные и методы функций*» знакомит читателя с языком TypeScript, начиная с базовых типов и их аннотаций, а затем переходит к обсуждению переменных, функций и расширенных возможностей языка.

`Глава 3` «*Интерфейсы, классы и наследование*» основана на работе из предыдущей главы и знакомит с объектно-ориентированными концепциями и возможностями интерфейсов, классов и наследования и затем показывает эти концепции в работе через шаблон проектирования Factory.

`В главе 4` «*Декораторы, обобщения и асинхронные функции*» обсуждаются более продвинутые языковые возможности декораторов и обобщений, прежде чем приступить к понятиям асинхронного программирования. Здесь показано, как язык TypeScript поддерживает эти асинхронные функции с помощью промисов и использования конструкций `async await`.

`Глава 5` «*Файлы объявлений и строгие опции компилятора*» рассказывает читателю о создании файла объявлений для существующего тела кода JavaScript, а затем перечисляет некоторые из наиболее распространенных синтаксисов, используемых при написании файлов объявлений, в виде шпаргалки. Затем обсуждаются строгие настройки компилятора, доступные для него, где они должны использоваться и какие преимущества дают.

`Глава 6` «*Сторонние библиотеки*» рассказывает читателю, как использовать файлы объявлений из репозитория DefinLYTyped в среде разработки, а затем показывает, как написать код на TypeScript, совместимый с тремя популярными фреймворками JavaScript – Backbone, AngularJS (версия 1) и ExtJS.

В `главе 7` «*Фреймворки, совместимые с TypeScript*» рассматриваются популярные фреймворки, которые полностью интегрированы в язык TypeScript. В ней исследуется парадигма MVC, а затем сравнивается, как данный шаблон проектирования реализован в Backbone, Aurelia, Angular 2 и React. Пример программы, которая использует ввод на основе форм, реализован в каждом их этих фреймворков.

Глава 8 «*Разработка через тестирование*» начинается с обсуждения того, что такое разработка через тестирование, а затем читатель проходит через процесс создания различных типов модульных тестов. Используя библиотеку Jasmine, вы увидите, как применять управляемые данными тесты и как тестировать асинхронную логику. Глава заканчивается обсуждением модулей, выполняющих тестирование, составлением отчетов о тестировании и использованием серверов непрерывной интеграции.

В главе 9 «*Тестирование фреймворков, совместимых с TypeScript*» показано, как протестировать пример приложения, созданного с использованием каждого из совместимых с TypeScript фреймворков. Здесь стратегия тестирования разбивается на тесты модели, тесты представления и тесты контроллера и показаны различия между стратегиями тестирования этих платформ.

В главе 10 «*Модуляризация*» рассказывается, что такое модули, как их можно использовать и о двух типах генерации модулей, которые поддерживает компилятор TypeScript: CommonJS и AMD, после чего показано, как можно применять модули с загрузчиками модулей, включая Require и SystemJS. Затем в этой главе подробно рассматривается использование модулей в Node для создания приложения Express. Наконец, обсуждается использование модулей в бессерверной среде при помощи функций AWS Lambda.

В главе 11 «*Объектно-ориентированное программирование*» обсуждаются концепции объектно-ориентированного программирования, а затем показано, как упорядочить компоненты приложения в соответствии с принципами ООП. Затем подробно рассматривается реализация передовых объектно-ориентированных практик, показывающих, как можно использовать шаблоны проектирования State и Mediator для управления сложными взаимодействиями пользовательского интерфейса.

В главе 12 «*Внедрение зависимости*» рассматриваются концепции размещения служб и внедрения зависимостей, а также способы их использования для решения типичных проблем разработки приложений. Затем показано, как реализовать простой фреймворк внедрения зависимостей, используя декораторы.

В главе 13 «*Создание приложений*» исследуются основные строительные блоки разработки веб-приложений, показывая, как интегрировать сервер Express и сайт на Angular, а также все важные механизмы авторизации, которые должен иметь любой сайт, с подробным обсуждением JWT-токенов. Наконец, в этой главе показано, как интегрировать социальный логин (способ авторизации с использованием существующего аккаунта в социальных сетях, таких как Google или Facebook).

В главе 14 «*Переходим к практике*» создается одностраничное приложение с использованием Angular и Express, объединяя все концепции и компоненты, собранные в книге, в одно приложение. Эти концепции включают в себя разработ-

ку через тестирование, государство и шаблоны State и Mediator, проектирование и использование конечных точек Express REST, принципы объектно-ориентированного проектирования и модуляризацию. В этой главе также рассматриваются общие методы при использовании наблюдаемых для обработки большинства типов взаимодействия REST API.

Чтобы получить максимальную отдачу от этой книги

Вам понадобится компилятор TypeScript и какой-нибудь редактор. Компилятор TypeScript доступен для Windows, MacOS и Linux в качестве плагина Node. В главе 1 *«Инструменты TypeScript и параметры фреймворков»* описана настройка среды разработки.

Загрузите файлы с примерами кода

Вы можете загрузить файлы с примерами кода для этой книги, используя свою учетную запись на сайте `www.packt.com`. Если вы приобрели эту книгу в другом месте, то можете посетить веб-сайт `www.packt.com/support` и зарегистрироваться, чтобы получить файлы по электронной почте.

Можно загрузить файлы с кодом, выполнив следующие действия.

1. Войдите или зарегистрируйтесь на сайте `www.packt.com`.
2. Выберите вкладку **ПОДДЕРЖКА**.
3. Нажмите **Загрузки кода и Ошибки**.
4. Введите название книги в поле поиска и следуйте инструкциям на экране.

Как только файл будет загружен, пожалуйста, убедитесь, что вы распаковали или извлекли папку, используя последнюю версию:

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

Пакет кода для книги также размещен на GitHub по адресу `https://github.com/PacktPublishing/Mastering-TypeScript-3`. В случае обновления кода он будет обновлен в существующем репозитории GitHub.

У нас также есть другие пакеты кода из нашего богатого каталога книг и видео, доступных на `https://github.com/PacktPublishing`. Посмотрите их!

Скачать цветные изображения

Мы также предоставляем файл PDF с цветными изображениями скриншотов/диаграмм, используемых в этой книге. Вы можете скачать его по адресу: http://dmkpress.com.

Используемые условные обозначения

В этой книге используется ряд текстовых обозначений.

`Текст кода`: указывает кодовые слова в тексте, имена таблиц базы данных, имена папок, имена файлов, расширения файлов, пути, фиктивные URL-адреса, пользовательский ввод и маркеры Twitter. Пример: «Этот файл `gruntfile.js` необходим для настройки всех задач Grunt».

Блок кода выглядит следующим образом:

```
test = this is a string
test = 1
test = function (a, b) {
   return a + b;
}
```

Когда мы хотим обратить ваше внимание на определенную часть блока кода, соответствующие строки или элементы выделяются жирным шрифтом:

```
declare function describe(
   description: string,
   specDefinitions: () => void
): void;
```

Любой ввод или вывод командной строки записывается следующим образом:

```
npm install -g typcript
```

Bold: указывает на новый термин, важное слово или слова, которые вы видите на экране. Например, слова в меню или диалоговых окнах появляются в тексте следующим образом. Вот пример: «Выберите **Системная информация** на панели **администрирования**».

Предупреждения или важные заметки выглядят так.

Предисловие от издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны. Вы можете написать отзыв прямо на нашем сайте www.dmkpress.com, зайдя на страницу книги, и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу **dmkpress@gmail.com**, при этом напишите название книги в теме письма. Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написа-нии новой книги, заполните форму на нашем сайте по адресу **http://dmkpress.com/authors/publish_book/** или напишите в издательство по адресу **dmkpress@gmail.com**.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг — возможно, ошибку в тексте или в коде, — мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии этой книги. Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу **dmkpress@gmail.com**, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Packt очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли применить санкции.Пожалуйста, свяжитесь с нами по адресу электронной почты **dmkpress@gmail.com** со ссылкой на подозрительные материалы.Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Глава 1

Инструменты TypeScript и параметры фреймворков

JavaScript – это поистине вездесущий язык: чем больше вы смотрите, тем больше JavaScript работает в самых неожиданных местах. Почти каждый веб-сайт, который вы посещаете в современном мире, использует JavaScript, чтобы сделать сайт более отзывчивым, более читабельным или более привлекательным в использовании. Даже традиционные настольные приложения выходят в сеть. Там, где нам когда-то требовалось скачать и установить программу для создания диаграммы или написания документа, теперь у нас есть возможность делать все это в интернете, в рамках своего скромного браузера.

Это сила JavaScript. Она позволяет нам переосмыслить способ использования интернета, но также позволяет переосмыслить то, как мы используем веб-технологии. Например, Node дает возможность JavaScript запускаться на стороне сервера, предоставляя целые крупномасштабные веб-сайты с обработкой сеансов, балансировкой нагрузки и взаимодействием с базой данных. Однако эта перемена в концепции относительно веб-технологий – только начало.

Apache Cordova – это полноценный веб-сервер, который работает как собственное приложение для мобильных телефонов. Это означает, что мы можем создать приложение для мобильного телефона с использованием HTML, CSS и JavaScript, а затем взаимодействовать с акселерометром телефона, службами геолокации или хранилищем файлов. Таким образом, благодаря Cordova JavaScript и веб-технологии переместились в сферу собственных приложений для мобильных телефонов.

Аналогичным образом такие проекты, как *Kinoma*, используют JavaScript для управления устройствами для **интернета вещей** (**IoT**), работающими на крошечных микропроцессорах, встроенных во все виды устройств. **Espruino** – чип микроконтроллера, специально предназначенный для запуска JavaScript. Таким образом, JavaScript теперь может управлять микропроцессорами на встроенных устройствах.

Настольные приложения также могут быть написаны на JavaScript и взаимодействовать с файловой системой с помощью таких проектов, как *Electron*. Это позволяет выполнить однократную запись и последующий запуск в любой операционной системе из коробки. На самом деле два самых популярных редактора исходного кода, Atom и Visual Studio Code, были созданы с использованием *Electron*.

Поэтому изучение JavaScript означает, что у вас есть возможность создавать веб-сайты, приложения для мобильных телефонов, настольные приложения и приложения IoT для запуска на встроенных устройствах.

Язык JavaScript не сложен для изучения, но он создает проблемы при написании больших и непростых программ, особенно в команде, работающей над одним проектом. Одна из основных проблем заключается в том, что JavaScript является интерпретируемым языком, а следовательно, у него нет шага компиляции. Проверка всего написанного кода на предмет мелких ошибок означает, что вы должны запустить его в интерпретаторе. Также традиционно было трудно реализовать принципы ООП в исходном формате на языке, а для создания подходящего, удобного в сопровождении и понятного кода на JavaScript требуется большая осторожность и дисциплина. Для программистов, которые переходят с других сильно типизированных объектно-ориентированных языков, таких как Java, C # или C ++, JavaScript может показаться совершенно чужой средой, особенно при работе с его более ранними версиями.

TypeScript устраняет этот пробел. Это сильно типизированный объектно-ориентированный язык, использующий компилятор для генерации JavaScript. Компилятор будет выявлять ошибки в кодовой базе еще до запуска в интерпретаторе. TypeScript также позволяет использовать хорошо известные методы ООП и шаблоны проектирования для создания приложений на JavaScript. Имейте в виду, что сгенерированный JavaScript – это просто JavaScript, и поэтому он будет работать везде, где может работать JavaScript, – в браузере, на сервере, мобильном устройстве, на рабочем столе или даже во встроенном устройстве.

Эта глава состоит из двух основных разделов. Первый раздел представляет собой краткий обзор ряда преимуществ использования TypeScript, а второй раздел посвящен настройке среды разработки TypeScript.

Если вы опытный программист на TypeScript и уже настроили среду разработки, можете пропустить эту главу. Если вы никогда прежде не работали с TypeScript и выбрали эту книгу, потому что хотите понять, что может делать TypeScript, читайте дальше.

В этой главе мы рассмотрим следующие темы.

Преимущества TypeScript:

- компиляция;
- сильная типизация;
- интеграция с популярными библиотеками JavaScript;
- инкапсуляция;• методы доступа public и private.

Настройка среды разработки:

- компиляция на основе Node;

- Visual Studio Code;
- Visual Studio 2017;
- WebStorm;
- другие редакторы и Grunt.

Что такое TypeScript?

TypeScript – это язык программирования, разработанный *Андерсом Хейлсбергом* (Anders Hejlsberg), создателем языка C #. Это результат оценки языка JavaScript и того, что можно сделать, чтобы помочь разработчикам при написании кода на JavaScript. TypeScript включает в себя компилятор, который преобразует код, написанный на TypeScript, в JavaScript. Его красота – в своей простоте. Мы можем взять существующий JavaScript, добавить несколько ключевых слов TypeScript тут и там и преобразовать свой код в сильно типизированную объектно-ориентированную кодовую базу с проверкой синтаксиса. Добавив шаг компиляции, мы можем проверить, что написали надежный код, который будет вести себя так, как мы хотели.

TypeScript генерирует JavaScript – все так просто. Это означает, что везде, где может использоваться JavaScript, TypeScript может применяться для генерации того же JavaScript, но с использованием проверок во время компиляции, чтобы гарантировать, что не нарушены определенные правила. Наличие этих дополнительных проверок еще до того, как мы запустим JavaScript, значительно экономит время, особенно там, где существуют большие команды разработчиков или где полученный код JavaScript публикуется как библиотека.

TypeScript также включает в себя языковую службу, которая может использоваться такими инструментами, как редакторы кода, чтобы помочь понять, как следует применять функции и библиотеки JavaScript. Эти редакторы могут затем автоматически предоставить программисту подсказки кода и советы по использованию данных библиотек.

Язык TypeScript, его компилятор и связанные с ним инструменты помогают разработчикам на JavaScript работать более продуктивно, быстрее находить ошибки и помогать друг другу понять, как использовать их код. Это дает нам возможность использовать протестированные концепции ООП и шаблоны проектирования в нашем коде JavaScript в очень простой и понятной форме. Давайте попробуем понять, как это происходит.

JavaScript и ECMAScript

JavaScript как язык существует уже давно. Первоначально разработанный как язык для поддержки HTML в одном веб-браузере, он вдохновил множество клонов языка, каждый со своими реализациями. В конце концов, был введен глобальный стандарт, позволяющий веб-сайтам поддерживать несколько браузеров. Язык, определенный в этом стандарте, называется **ECMAScript**.

Каждый интерпретатор JavaScript должен предоставлять функции и свойства, соответствующие стандарту ECMAScript. Стандарт ECMAScript, который был опубликован в 1999 году, официально назывался **ECMA-262**, **3-е издание**, но стал называться просто **ECMAScript 3**. Эта версия JavaScript получила широкое распространение и стала основой для взрывной популярности и роста интернета в том виде, в котором мы его знаем.

С популярностью языка и растущим использованием вне веб-браузера стандарт ECMAScript неоднократно пересматривался и обновлялся. К сожалению, время, которое требуется между предложением новых языковых функций и ратификацией нового стандарта для их покрытия, может быть довольно длительным. Даже когда публикуется новая версия стандарта, веб-браузеры принимают эти стандарты только с течением времени, а также могут реализовывать фрагменты стандарта раньше других.

Поэтому, прежде чем выбирать, какой стандарт принять, важно понять, какие браузеры или, точнее, какой механизм времени выполнения необходимо будет поддерживать. Чтобы поддержать эти решения, есть ряд справочных сайтов, которые перечисляют поддержку в так называемой таблице совместимости.

В настоящее время на выбор предлагается три основные версии ECMAScript: ES3, ES5 и недавно ратифицированная ES6. ES3 существует уже давно, и почти любой веб-браузер будет поддерживать ее. ES5 поддерживается большинством современных веб-браузеров. ES6 является последней версией стандарта и на сегодняшний день самым крупным обновлением языка до настоящего времени. Она впервые вводит в язык классы, облегчая реализацию объектно-ориентированного программирования.

Компилятор TypeScript имеет параметр, который может переключаться между различными версиями стандарта ECMAScript. В настоящее время TypeScript поддерживает ES3, ES5 и ES6. Когда компилятор запускает ваш TypeScript, он будет генерировать ошибки компиляции, если код, который вы пытаетесь компилировать, не соответствует этому стандарту. Команда Microsoft взяла на себя обязательство следовать стандартам ECMAScript в любых новых версиях компилятора TypeScript, поэтому по мере принятия новых редакций язык TypeScript и компилятор будут следовать их примеру.

Преимущества TypeScript

Чтобы получить представление о преимуществах TypeScript, давайте очень кратко рассмотрим некоторые моменты, которые TypeScript дает в таблице:

- компиляцию;
- сильную типизацию;
- интеграцию с популярными библиотеками JavaScript;
- инкапсуляцию;
- методы доступа public и private.

Компиляция

Одна из самых любимых черт JavaScript – отсутствие шага компиляции. Просто измените свой код, обновите браузер, а интерпретатор позаботится обо всем остальном. Нет необходимости ждать какое-то время, пока компилятор не закончит работу, чтобы запустить свой код.

Хотя можно рассматривать это как преимущество, есть много причин, по которым вы захотите использовать шаг компиляции. Компилятор может найти глупые ошибки, такие как пропущенные скобки или запятые. Он также может найти другие более невнятные ошибки, такие как использование одной кавычки (') там, где должна использоваться двойная кавычка ("). Каждый разработчик JavaScript может рассказать страшные истории о часах, потраченных на поиск ошибок в своем коде, только для того, чтобы узнать, что он пропустил случайную закрывающую скобку } или простую запятую.

Использование шага компиляции в своем рабочем процессе действительно упрощает работу при управлении большой базой кода. Существует старая поговорка, в которой говорится, что мы должны рано выходить из строя и громко выходить из строя, а компилятор будет очень громко кричать на самой ранней возможной стадии, когда будут обнаружены ошибки. Это означает, что любая фиксация изменений в исходном коде будет свободна от ошибок, обнаруженных компилятором.

При внесении изменений в большую кодовую базу нам также необходимо убедиться, что мы не нарушаем существующую функциональность. В большой команде это часто означает использование ветвления и слияния репозитория исходного кода. Выполнение шага компиляции до, во время и после слияния из одной ветки в другую дает нам дополнительную уверенность в том, что мы не совершили ошибок или что в процессе автоматического слияния также не было допущено ошибок.

Если команда разработчиков использует процесс непрерывной интеграции, сервер **непрерывной интеграции** (**CI**) может быть ответственным за создание и развертывание всего сайта, а затем за выполнение набора модульных и инте-

грационных тестов для вновь зарегистрированного кода. Мы можем сэкономить часы, которые уходят на сборку и тестирование, гарантируя отсутствие синтаксических ошибок в коде, прежде чем приступать к развертыванию и запуску тестов.

Наконец, как упоминалось ранее, компилятор TypeScript может быть настроен на вывод ES3, ES5 или ES6 JavaScript. Это означает, что мы можем ориентироваться на разные версии среды выполнения из одной и той же базы кода.

Сильная типизация

JavaScript не сильно типизирован. Это очень динамичный язык, поскольку он позволяет объектам изменять свои свойства и поведение на лету. В качестве примера рассмотрим приведенный ниже код:

```
var test = "this is a string;
console.log('test=' + test);

test = 1;
console.log('test=' + test);

test = function (a, b) {
   return a + b;
}

console.log('test=' + test);
```

В первой строке этого фрагмента объявлена переменная с именем `test` и ей присвоено строковое значение. Чтобы убедиться в этом, мы записали значение в консоль. Затем мы присваиваем числовое значение переменной `test` и снова записываем ее значение в консоль. Обратите внимание, однако, на последний фрагмент кода. Мы назначаем функцию, которая принимает два параметра для переменной `test`. Если мы запустим этот код, то получим следующие результаты:

```
test = this is a string
test = 1
test = function (a, b) {
   return a + b;
}
```

Здесь ясно видны изменения, которые мы вносим в переменную `test`. Она меняется, переходя от строкового значения к числовому, а затем становится функцией.

Изменение типа переменной во время выполнения может быть очень опасным занятием и привести к бесчисленным проблемам. Вот почему традиционные объектно-ориентированные языки обеспечивают сильную типизацию. Другими словами, они не позволяют природе переменной изменяться после объявления.

Хотя весь приведенный выше код является допустимым кодом JavaScript – и может быть оправдан, – довольно легко видно, как это может привести к ошибкам во время выполнения. Представьте, что вы были ответственны за написание библиотечной функции для добавления двух чисел, а затем еще одного разработчика, который непреднамеренно переназначил вашу функцию, вместо того чтобы вычитать эти числа.

Подобного рода ошибки можно легко обнаружить в нескольких строках кода, но находить и исправлять их по мере расширения вашей базы кода и команды разработчиков будет все труднее.

Синтаксический сахар

TypeScript представляет очень простой синтаксис для проверки типа объекта во время компиляции. Этот синтаксис известен как синтаксический сахар, или, более формально, аннотация типов. Рассмотрим приведенную ниже версию нашего исходного кода JavaScript, написанного на TypeScript:

```
var test: string = "this is a string";
test = 1;
test = function(a, b) { return a + b; }
```

Обратите внимание, что в первой строке этого фрагмента мы ввели двоеточие `:` и ключевое слово `string` между нашей переменной и ее присваиванием. Этот синтаксис аннотации типа означает, что мы устанавливаем тип нашей переменной как тип `string` и что любой код, который не придерживается этих правил, приведет к ошибке компиляции. Запуск предыдущего кода через компилятор TypeScript вызовет две ошибки:

```
hello.ts(3,1): error TS2322: Type 'number' is not assignable to
type 'string'.
hello.ts(4,1): error TS2322: Type '(a: any, b: any) => any' is
not assignable to type 'string'.
```

Первая ошибка довольно очевидна. Мы указали, что переменная `test` является строкой, и поэтому попытка присвоить ей номер вызовет ошибку компиляции. Вторая ошибка похожа на первую и, по сути, говорит о том, что мы не можем присвоить функцию строке.

Таким образом, компилятор TypeScript вводит сильную или статическую типизацию в нашем коде JavaScript, предоставляя нам все преимущества сильно типизированного языка. Поэтому TypeScript описывается как расширенный вариант JavaScript. Мы рассмотрим это более подробно в главе 2 *«Типы, переменные и методы функций»*.

Определение типов для популярных библиотек JavaScript

Как мы уже видели, TypeScript имеет возможность аннотировать JavaScript и вносить сильную типизацию в опыт разработки на JavaScript. Но как можно сильно типизировать существующие библиотеки JavaScript? Ответ на удивление прост: путем создания файла определения. TypeScript использует файлы с расширением `.d.ts` как своего рода заголовочный файл, по аналогии с такими языками, как C ++, для наложения сильной типизации на существующие библиотеки JavaScript. Эти файлы определений содержат информацию, которая описывает каждую доступную функцию и/или переменные, а также связанные с ними аннотации типов.

Давайте быстро посмотрим, как будет выглядеть определение. В качестве примера рассмотрим функцию из популярного фреймворка для модульного тестирования Jasmine под названием `describe`:

```
var describe = function(description, specDefinitions) {
   return jasmine.getEnv().describe(description, specDefinitions);
};
```

Обратите внимание на то, что у функции `describe` есть два параметра – `description` и `specDefinitions`. Но JavaScript не говорит нам, что это за переменные. Нам нужно взглянуть на документацию по Jasmine, чтобы выяснить, как вызвать эту функцию: если мы перейдем на страницу `http://jasmine.GitHub.io/2.0/introduction.html`, то увидим пример использования этой функции:

```
describe("A suite", function () {
   it("contains spec with an expectation", function () {
      expect(true).toBe(true);
   });
});
```

Таким образом, из документации явствует, что первый параметр является строкой, а второй – это функция. Но в JavaScript нет ничего, что заставляло бы нас соответствовать этому API. Как упоминалось ранее, мы могли бы легко вызвать эту функцию с двумя числами или непреднамеренно переключить параметры, отправив сначала функцию, а затем строку. Очевидно, мы начнем получать ошибки среды выполнения, если сделаем это, но TypeScript, используя файл определений, может генерировать ошибки времени компиляции, прежде чем мы попытаемся запустить этот код.

Давайте посмотрим на фрагмент файла определения `jasmine.d.ts`:

```
declare function describe(
   description: string,
```

```
    specDefinitions: () => void
): void;
```

Это определение TypeScript для функции `describe`. Во-первых, `declare function describe` говорит нам, что мы можем использовать функцию под названием describe, но реализация этой функции будет обеспечиваться в среде выполнения.

Ясно, что параметр `description` сильно типизирован, чтобы стать строкой (`string`), а параметр `specDefinitions` сильно типизирован, чтобы стать функцией (`function`), которая возвращает `void`. TypeScript использует двойные скобки `()` для объявления функций и стрелку для отображения типа возвращаемого значения функции. Следовательно, `() => void` – это функция, которая ничего не возвращает. В конечном итоге функция describe сама вернет void.

Представьте, что наш код должен попытаться передать функцию в качестве первого параметра и строку в качестве второго параметра (явно нарушая определение этой функции), как показано в приведенном ниже примере:

```
describe(() => { /* function body */}, "description");
```

В этом случае TypeScript выдаст следующую ошибку:

```
hello.ts(11,11): error TS2345: Argument of type '() => void'
is not assignable to parameter of type 'string'.
```

Эта ошибка говорит о том, что мы пытаемся вызвать функцию describe с недопустимыми параметрами. Мы рассмотрим файлы определений более подробно в последующих главах, но этот пример ясно показывает, что TypeScript будет генерировать ошибки, если мы попытаемся использовать внешние библиотеки JavaScript неправильно.

DefinitelyTyped

Вскоре после выхода TypeScript *Борис Янков* запустил GitHub-репозиторий для размещения файлов определений под названием DefiniteTyped (`http://definitetyped.org`). Этот репозиторий стал первым портом захода для интеграции внешних библиотек в TypeScript и в настоящее время содержит определения для более чем 1600 библиотек JavaScript. Рост этого сайта и скорость создания типов определений для множества библиотек JavaScript показывают популярность TypeScript.

Инкапсуляция

Одним из фундаментальных принципов объектно-ориентированного программирования является инкапсуляция, возможность определять данные, а также набор функций, которые могут работать с этими данными, в единый компонент.

Большинство языков программирования обладает концепцией класса для этой цели – предоставляя способ определения шаблона для данных и связанных функций.

Для начала давайте посмотрим на простое определение класса TypeScript:

```
class MyClass {
   add(x, y) {
      return x + y;
   }
}

var classInstance = new MyClass();
var result = classInstance.add(1,2);
console.log('add(1,2) returns ${result}');
```

Этот код довольно прост для чтения и понимания. Мы создали класс `MyClass` с простой функцией `add`. Чтобы использовать его, мы просто создаем его экземпляр класса и вызываем функцию `add` с двумя аргументами.

JavaScript, предшествующий ES6, не имеет оператора класса, но вместо этого использует функции для воспроизведения функциональности классов. Инкапсуляция через классы достигается с помощью либо шаблона прототипа, либо шаблона замыкания. Понимание прототипов и шаблона замыкания и их правильное использование считаются фундаментальным навыком при написании кода на JavaScript корпоративного уровня.

Замыкание – это, по сути, функция, которая ссылается на независимые переменные. Это означает, что переменные, определенные в функции замыкания, запоминают среду, в которой они были созданы. Это позволяет JavaScript определять локальные переменные и предоставлять инкапсуляцию. Запись определения `MyClass` в предыдущем коде с использованием замыкания в JavaScript будет выглядеть примерно так:

```
var MyClass = (function () {
   // самовызывающаяся функция - это среда,
   // которую замыкание запомнит;
   function MyClass() {
      // MyClass - это внутренняя функция,
      // замыкание;
   }
      MyClass.prototype.add = function (x, y) {
      return x + y;
   };
   return MyClass;
})();
```

```
var classInstance = new MyClass();
var result = classInstance.add(1, 2);
console.log("add(1,2) returns " + result);
```

Мы начнем с переменной `MyClass` и назначим ее функции, которая выполняется немедленно, – обратите внимание на `}) ();` синтаксис в нижней части определения замыкания. Этот синтаксис является распространенным способом написания кода на JavaScript во избежание утечки переменных в глобальное пространство имен. Затем мы определяем новую функцию с именем `MyClass` и возвращаем ее внешней вызывающей функции. После этого мы используем ключевое слово `prototype`, чтобы добавить новую функцию в определение `MyClass`. Эта функция называется `add` и принимает два параметра, возвращая их сумму.

Последние несколько строк предыдущего фрагмента кода показывают, как использовать это замыкание в JavaScript. Создайте экземпляр типа замыкания, а затем выполните функцию add. Выполнение этого кода будет регистрировать **`add(1,2) returns 3`** в консоль, как и ожидалось.

Сравнив код JavaScript с кодом TypeScript, легко увидеть, насколько просто выглядит TypeScript по сравнению с эквивалентным JavaScript. Помните, мы упоминали, что программисты на JavaScript могут легко потерять фигурную `{` или обычную `(` скобку? Посмотрите на последнюю строку в определении замыкания: `})();` на обнаружение одной такой скобки может уйти несколько часов отладки.

Классы TypeScript генерируют замыкания

JavaScript, как показано ранее, на самом деле является результатом определения класса TypeScript. Итак, TypeScript действительно генерирует замыкания за вас.

О добавлении концепции классов в язык JavaScript говорилось годами, и в настоящее время она является частью **ECMAScript 6th Edition**. Microsoft взяла на себя обязательство следовать стандарту ECMAScript в компиляторе TypeScript, как и когда эти стандарты публикуются.

Методы доступа public и private

Еще один принцип ООП, который используется в инкапсуляции, – это концепция сокрытия данных, то есть способность иметь открытые и закрытые переменные. Закрытые переменные должны быть скрыты для пользователя определенного класса, поскольку должны использоваться только самим классом. Случайное раскрытие этих переменных может легко привести к ошибкам во время выполнения.

К сожалению, в JavaScript нет встроенного способа объявления переменных закрытыми. Хотя эту функциональность можно эмулировать с помощью замыканий, многие программисты на JavaScript просто используют символ подчеркива-

ния (_) для обозначения закрытой переменной. Хотя во время выполнения, если вы знаете имя закрытой переменной, вы можете легко присвоить ей значение. Рассмотрим приведенный ниже код:

```
var MyClass = (function() {
   function MyClass() {
      this._count = 0;
   }
   MyClass.prototype.countUp = function() {
      this._count ++;
   }
   MyClass.prototype.getCountUp = function() {
      return this._count;
   }
   return MyClass;
}());var test = new MyClass();
test._count = 17;
console.log("countUp : " + test.getCountUp());
```

Переменная `MyClass` на самом деле является замыканием с функцией-конструктором, функцией `countUp` и функцией `getCountUp`. Предполагается, что переменная `_count` является закрытой переменной-членом, которая используется только в рамках замыкания. Использование соглашения об именах подчеркивания дает пользователю этого класса некоторое представление о том, что переменная является закрытой, но JavaScript все равно позволит вам манипулировать переменной `_count`. Посмотрите на вторую последнюю строку фрагмента кода. Мы явно устанавливаем значение `_count` равным 17, что разрешено JavaScript, но не желательно для автора класса. Результатом этого кода будет **`countUp: 17`**.

TypeScript, однако, использует ключевые слова `public` и `private`, которые можно использовать для переменных членов класса. Попытка получить доступ к переменной члена класса, которая была помечена как private, приведет к ошибке времени компиляции. Как пример этого предыдущий код может быть написан на TypeScript следующим образом:

```
class CountClass {
   private _count: number;
   constructor() {
      this._count = 0;
   }
   countUp() {
      this._count ++;
   }
   getCount() {
      return this._count;
   }
}
```

```
var countInstance = new CountClass() ;
countInstance._count = 17;
```

Здесь, во второй строке нашего фрагмента кода, мы объявили закрытую переменную-член с именем `_count`. Опять же, у нас есть конструктор, функция `countUp` и функция `getCount`. Если мы скомпилируем этот файл, компилятор выдаст ошибку:

```
hello.ts(39,15): error TS2341: Property '_count' is private and
only accessible within class 'CountClass'.
```

Эта ошибка появляется, потому что мы пытаемся получить доступ к закрытой переменной `_count` в последней строке кода.

Поэтому компилятор TypeScript помогает нам придерживаться открытых и закрытых методов доступа, выдавая ошибку компиляции, когда мы непреднамеренно нарушаем данное правило.

Помните, однако, что эти методы доступа являются всего лишь функцией времени компиляции и не влияют на сгенерированный код JavaScript. Вам необходимо помнить об этом, если вы пишете библиотеки JavaScript, которые будут использоваться третьими сторонами. Обратите внимание, что по умолчанию компилятор TypeScript по-прежнему генерирует выходной файл JavaScript, даже при наличии ошибок компиляции. Однако этот параметр можно изменить, чтобы компилятор TypeScript не генерировал JavaScript, если есть ошибки компиляции.

Интегрированные среды разработки TypeScript

Цель данного раздела – научить вас работать со средой TypeScript, чтобы вы могли редактировать, компилировать, запускать и отлаживать свой код, написанный на TypeScript. TypeScript был выпущен как проект с открытым исходным кодом и включает в себя как вариант для Windows, так и вариант для Node. Это означает, что компилятор будет работать в Windows, Linux, macOS и любой другой операционной системе, которая поддерживает Node. В средах Windows можно установить Visual Studio, которая зарегистрирует `tsc.exe` (компилятор TypeScript) в нашем каталоге `c:\Program Files`, либо можно использовать Node. В средах Linux и macOS нам нужно будет использовать Node.

В этом разделе мы рассмотрим следующие среды разработки:

- компиляцию на основе Node;
- Visual Studio Code;

- Visual Studio 2017;
- WebStorm;
- использование Grunt.

Компиляция на основе Node

Самая простая среда разработки TypeScript состоит из простого текстового редактора и компилятора TypeScript на основе Node. Перейдите на сайт Node (`https://nodejs.org`) и следуйте инструкциям по установке Node на выбранной вами операционной системе.

Как только Node будет установлен, можно установить TypeScript, просто набрав:

```
npm install -g typescript
```

Эта команда вызывает диспетчер пакетов Node (`npm`) для установки TypeScript в качестве глобального модуля (опция `-g`), который сделает его доступным независимо от того, в каком каталоге мы сейчас находимся. После установки TypeScript мы можем отобразить текущую версию компилятора, набрав следующее:

```
tsc -v
```

На момент написания этой главы версия компилятора TypeScript – 3.3.3, и поэтому вывод этой команды такой:

```
Version 3.3.3
```

Давайте теперь создадим файл TypeScript с именем `hello.ts` со следующим содержимым:

```
console.log('hello TypeScript');
```

Из командной строки мы можем использовать TypeScript для компиляции этого файла в файл JavaScript, выполнив следующую команду:

```
tsc hello.ts
```

Как только компилятор TypeScript завершит свою работу, он сгенерирует файл `hello.js` в текущем каталоге. Мы можем запустить этот файл с помощью Node, набрав:

```
node hello.js
```

После этого в консоль будет выведено:

```
hello TypeScript
```

Создание файла tsconfig.json

Компилятор TypeScript использует файл tsconfig.json в корне каталога проекта для указания любых глобальных параметров проекта TypeScript и параметров компилятора. Это означает, что вместо компиляции своих файлов TypeScript один за другим (указав каждый файл в командной строке) мы можем просто набрать tsc из корневого каталога проекта, и TypeScript рекурсивно найдет и скомпилирует все файлы TypeScript в корневом каталоге и во всех подкаталогах. Файл `tsconfig.json`, который необходим TypeScript для этого, можно создать из командной строки, просто набрав:

```
tsc -init
```

Результатом этой команды является основной файл `tsconfig.json`:

```
{
   "compilerOptions": {
      "target": "es5",
      "module": "commonjs",
      "strict": true,
      "esModuleInterop": true
   }
}
```

Это простой файл формата JSON с единственным свойством `compilerOptions`, которое определяет параметры компиляции для проекта. Свойство target указывает предпочтительный вывод JavaScript для генерации, и это может быть либо `es3`, `es5`, `es6`, `ES2016`, `ES2017`, либо `ESNext`. Параметр `strict` – это флаг, который включает все параметры строгой проверки типов. Мы рассмотрим значение этих параметров в `главе 5` *«Файлы объявлений и строгие опции компиляции»*. Опция `esModuleInterop` связана с генерацией модулей, которую мы также обсудим в `главе 10` *«Модуляризация»*.

TypeScript позволяет использовать несколько файлов tsconfig.json в структуре каталога. Это дает возможность различным подкаталогам применять разные параметры компилятора.

Используя наш файл `tsconfig.json`, мы можем скомпилировать свое приложение, просто набрав:

```
tsc
```

Эта команда вызовет компилятор TypeScript, используя созданный нами файл `tsconfig.json`, и сгенерирует файл `hello.js`. Фактически любой исходный файл TypeScript с расширением `.ts` будет генерировать файл JavaScript с расширением `.js`.

Мы успешно создали простую среду разработки TypeScript на основе Node, с простым текстовым редактором и доступом к командной строке.

Локализованные сообщения

При вызове компилятора TypeScript через командную строку мы также можем указать, что любые сообщения должны отображаться на определенном языке. Это выполняется с помощью параметра командной строки `--locale` следующим образом:

```
tsc --locale pl
```

Здесь мы компилируем наш исходный код как обычно, но вывод сообщения будет на польском, как показано ниже:

```
error TS2322: Typu "number" nie można przypisać do typu "string"
```

На момент написания этой главы возможные значения, которые можно использовать для опции компиляции `--locale`, следующие:

```
en English US
cs Czech
de German
es Spanish
fr French
it Italian
ja Japanese
ko Korean
pl Polish
ru Russian
tr Turkish
```

Для параметра `--a` нет соответствующей опции `tsconfig.json`, поэтому ее нужно будет включать при запуске tsc из командной строки.

При создании приложений TypeScript все фреймворки и вспомогательные инструменты, которые вам понадобятся, поставляются с интерфейсом командной строки. Нет ничего необычного в том, что разработчик запускает несколько окон командной строки, одно для автоматической компиляции кода, другое для запуска веб-сервера и еще одно для запуска модульных тестов, при изменении базы кода. Если вы разрабатываете для Windows, обратите внимание на отличный эмулятор командной строки `cmder (http://cmder.net)`. Он поддерживает несколько вкладок, разделенные экраны и включает в себя эмулятор Linux с готовой поддержкой git.

Если вы ищете хороший редактор исходного кода для TypeScript, обратите внимание на Visual Studio Code (VSCode). Использование среды командной строки

на основе Node и VSCode – это все, что вам нужно для создания любого проекта TypeScript в любой операционной системе. Если вы знакомы с Microsoft Visual Studio или WebStorm и предпочитаете полностью интегрированную среду разработки, эти две ИСР могут лучше всего подойти.

Visual Studio Code

Visual Studio Code (VSCode) – это легкая среда разработки, созданная Microsoft, которая работает на Windows, Linux и macOS. Она включает в себя функции разработки, такие как подсветка синтаксиса, сопоставление скобок, Intellisense, а также поддерживает множество разных языков: TypeScript, JavaScript, JSON, HTML, CSS, C#, C++, Java и многие другие, что делает ее идеальной для разработки на TypeScript с ориентацией на веб-страницы или Node. Основное внимание уделяется разработке Node с использованием TypeScript и ASP.NET. Основная разработка с C#. Она также имеет сильную поддержку Git из коробки и даже была создана с использованием TypeScript и Electron.

Установка VSCode

VSCode можно установить в Windows, просто загрузив и запустив установщик. Для систем Linux VSCode распространяется в виде пакета `.deb`, пакета `.rpm` или двоичного `tar`-файла. В macOS загрузите файл `.zip`, разархивируйте его, а затем скопируйте файл Visual Studio.Code.app в папку со своими приложениями.

Изучение VSCode

Создайте новый каталог для хранения своего исходного кода и запустите VSCode. Это можно сделать, перейдя в каталог и выполнив `code .` из командной строки. В системах Windows запустите VSCode, а затем выберите **File | Open folder** (Файл | Открыть папку) из строки меню. Нажмите сочетание клавиш *Ctrl+N*, чтобы создать новый файл, и введите:

```
console.log("hello vscode");
```

Обратите внимание, что на данном этапе нет подсветки синтаксиса, поскольку VSCode не знает, с каким типом файла он работает. Нажмите сочетание клавиш *Ctrl+S*, чтобы сохранить файл, и назовите его `hello.ts`. Теперь, когда VSCode понимает, что это файл TypeScript, у вас будет полный Intellisense и подстветка синтаксиса.

Создание файла tasks.json

VSCode использует специальный файл `tasks.json` для запуска часто используемых задач, которые могут вам понадобиться в жизненном цикле разработки.

Очевидно, что одной из наиболее распространенных задач, которые вам нужно будет выполнить, является шаг компиляции, также известный как этап сборки. Сочетание клавиш для сборки проекта в VSCode – *Ctrl+Shift+B*.

Чтобы выполнить этап сборки с помощью *Ctrl+Shift+B*, нам нужно будет создать файл `tasks.json` и правильно настроить его для запуска команды `tsc`, чтобы выполнить сборку своего проекта. Если у вас еще нет файла `tasks.json`, создайте его из командной строки, введя:

```
tsc --init
```

Теперь, когда у нас есть файл tsconfig.json для нашего проекта, мы можем настроить задачу сборки по умолчанию. В VSCode нажмите сочетание клавиш *Ctrl+Shift+B*, чтобы выбрать задачу сборки для запуска. Выберите опцию `tsc: build - tsconfig.json`. Будет выполнен шаг компиляции tsc для вашего проекта, а результаты будут отображены во встроенном терминале.

К сожалению, если мы снова нажмем *Ctrl+Shift+B*, нам будет предложено еще раз выбрать задачу сборки. Это означает, что хотя мы и настроили задачу сборки, мы не указали, какая задача должна запускаться по умолчанию, когда мы нажимаем на ярлык сборки. Есть два способа, чтобы указать эту задачу по умолчанию. Во-первых, выбрав пункт меню **Tasks | Configure Default build task** (**Задачи | Настроить задание сборки по умолчанию**). Появится опция, где мы можем выбрать задачу `tsc: build - tsconfig.json` по умолчанию. После выбора этой опции откроется файл tasks.json в каталоге .vscode в редакторе.

Второй способ указать задачу сборки по умолчанию – с помощью командной палитры. В VSCode есть много команд, которые могут быть настроены через командную палитру, и каждая из них может быть привязана к определенной комбинации клавиш. Выбрав пункт меню **View | Command Pallet** (**Вид | Командная палитра**) или нажав сочетание клавиш *Ctrl+Shift+P*, вы увидите список доступных команд. Этот список можно отфильтровать, просто набрав ключевое слово. Если мы начнем набирать слово task, список автоматически отфильтруется только для отображения команд, которые содержат слово «task». Затем мы можем выбрать **Tasks : configure default build task**, чтобы открыть файл `tasks.json` в редакторе.

Обратите внимание, что есть много доступных команд, которые можно выбрать с помощью пункта меню «**Вид | Командная палитра**» или нажав сочетание клавиш *Ctrl+Shift+P*. Потратьте несколько минут на то, чтобы просмотреть список команд, дабы увидеть, насколько конфигурируемым на самом деле является VSCode.

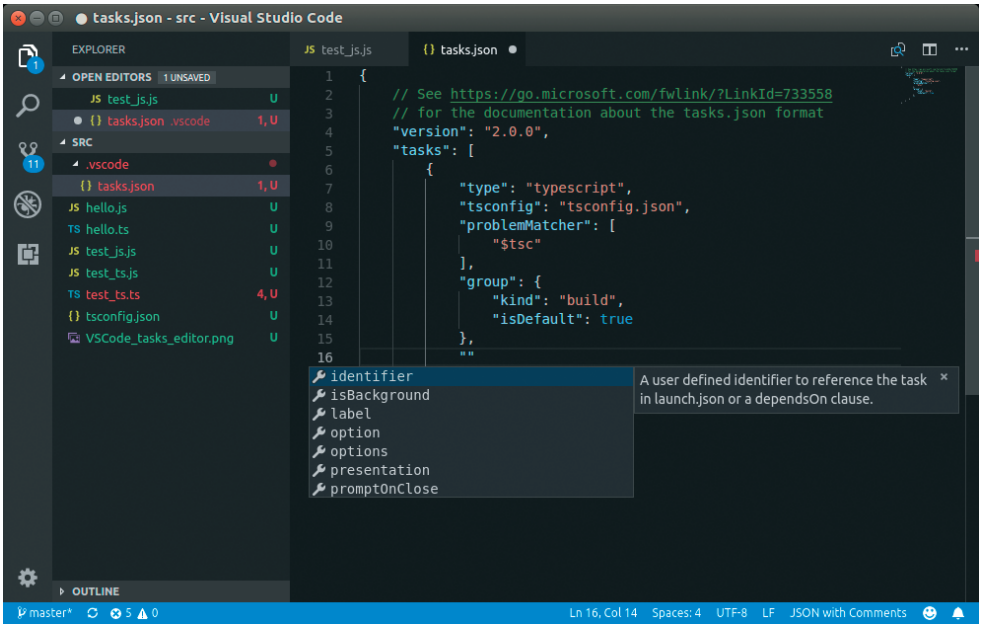
Независимо от того, перешли ли мы к конфигурируемым задачам сборки по умолчанию с помощью опции меню или из командной палитры, VSCode откроет наш файл tasks.json в редакторе следующим образом:

```
{
   // Посетите страницу https://go.Microsoft.com/wlink/
```

```
    // ?LinkId=733558 для получения документации
    // по формату tasks.json;
    "version": "2.0.0",
    "tasks": [
      {
        "type": "typescript",
        "tsconfig": "tsconfig.json",
        "problemMatcher": [
          "$tsc"
        ],
        "group": {
          "kind": "build",
          "isDefault": true
        }
      }
    ]
}
```

У этого файла `tasks.json` есть свойство `version` и свойство `tasks`. Свойство `tasks` – это массив задач, которые имеют ряд подсвойств. VSCode автоматически создал для нас задачу, которая скомпилирует наш проект, и указал с помощью свойства `group.isDefault`, что это задание по умолчанию.

Обратите внимание, что при редактировании данного файла VSCode автоматически показывает нам, какие свойства доступны в зависимости от того, какую часть файла мы модифицируем:



Здесь видно, что VSCode показывает нам, что доступные свойства конфигурации для задачи включают в себя `identifier`, `isBackground`, `label` и другие.

Удобное свойство, которое нужно иметь в виду, – это свойство `label`. Его можно использовать для указания легко запоминаемого имени для нашей задачи. Давайте добавим это свойство и установим его значение как `Run tsc build`. Теперь, когда мы видим эту задачу сборки в списке параметров, она будет отображаться как `Run tsc build`, что помогает нам различать задачи.

Сборка проекта

Наш пример проекта теперь можно собрать, нажав сочетание клавиш *Ctrl+Shift+B*. Обратите внимание, что в базовом каталоге нашего проекта у нас теперь есть файл `hello.js` и файл `hello.js.map` в результате шага компиляции.

Создание файла launch.json

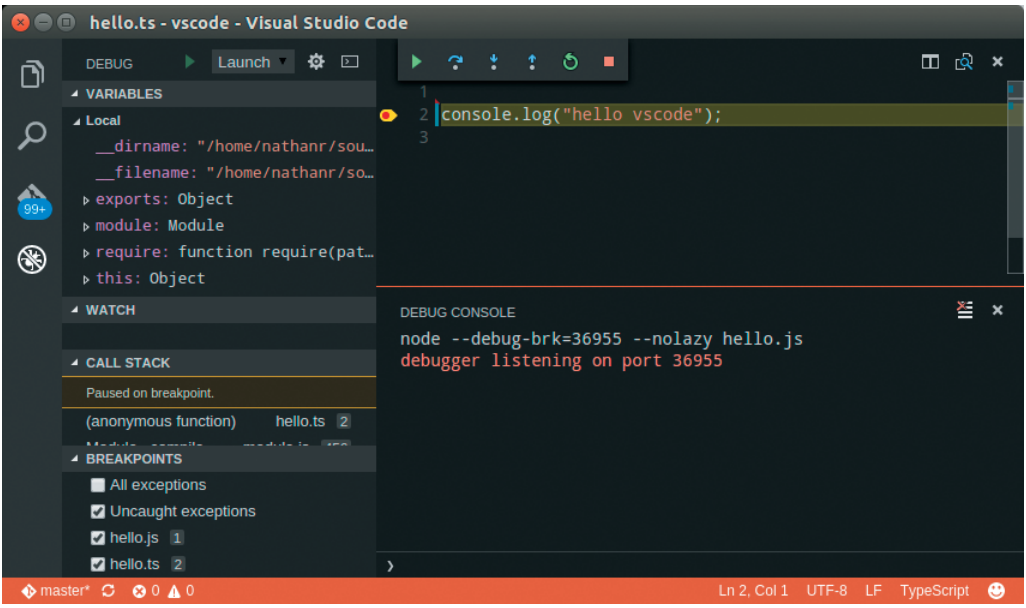
VSCode включает в себя встроенный отладчик, который можно использовать для отладки проектов TypeScript. Если мы переключимся на панель отладчика или просто нажмем клавишу *F5*, чтобы начать отладку, VSCode попросит нас выбрать среду отладки. В настоящее время выберите опцию **Node.js**, которая создаст файл `launch.json` в каталоге .vscode, и снова откройте его для редактирования. Найдите параметр с именем program и измените его на "`${workspaceFolder}/hello.js`". Нажмите клавишу *F5* еще раз, и VSCode запустит `hello.js` как программу Node и выведет результаты в окно отладки:

```
node --debug-brk=34146 --nolazy hello.js
debugger listening on port 34146
hello vscode
```

Установка точек останова

Использование точек останова и отладки на этом этапе будет работать только для сгенерированных файлов JavaScript `.js`. Нам потребуется внести изменения в файл `tsconfig.json`, чтобы включить отладку непосредственно в наших файлах TypeScript. Отредактируйте файл tsconfig.json и добавьте свойство с именем `sourceMaps` и значением свойства `true`. Оно указывает компилятору выводить исходный файл (с именем `.map`) для каждого файла TypeScript, который мы компилируем. Как только мы выполним повторную сборку проекта, эти файлы `.map` появятся в нашем дереве исходного кода.

Теперь мы можем установить точки останова непосредственно в наших файлах `.ts`, чтобы ими мог воспользоваться отладчик VSCode:



Отладка веб-страниц

Отладка TypeScript, которая выполняется на веб-странице в VSCode, требует немного больше настроек. VSCode использует отладчик Chrome для подключения к запущенной веб-странице. Чтобы включить отладку веб-страниц, нам сначала нужно добавить конфигурацию Debug в наш файл `launch.json`. К счастью, в VSCode для этого есть команда панели инструментов, и она сгенерирует конфигурацию запуска за нас. Выберите пункт меню **Debug | Add Configuration** (**Отладка | Добавить конфигурацию**), а затем параметр **Chrome Attach** (**Присоединить к Chrome**). Наш файл `launch.json` изменится и будет выглядеть так:

```
"configurations": [
   {
      "type": "chrome",
      "request": "attach",
      "name": "Attach to Chrome",
      "port": 9222,
      "webRoot": "${workspaceFolder}"
   },

   {
      "type": "node",
      ...
   }
]
```

Этот параметр запуска называется `Attach to Chrome` и будет подключаться к работающему экземпляру Chrome с использованием порта отладки `9222`. Сохраните файл `launch.json` и создайте HTML-страницу с именем `index.html` в корневом каталоге проекта:

```
<html>
   <head>
      <script src="helloweb.js"></script>
   </head>
   <body>
      hello vscode
      <div id="content"></div>
   </body>
</html>
```

Это очень простая страница, которая загружает файл `helloweb.js` и отображает текст `hello vscode`. Наш файл `helloweb.ts` выглядит следующим образом:

```
window.onload = () => {
   console.log("hello vscode");
};
```

Этот код TypeScript просто ожидает загрузки веб-страницы, а затем выводит `hello vscode` в консоль.

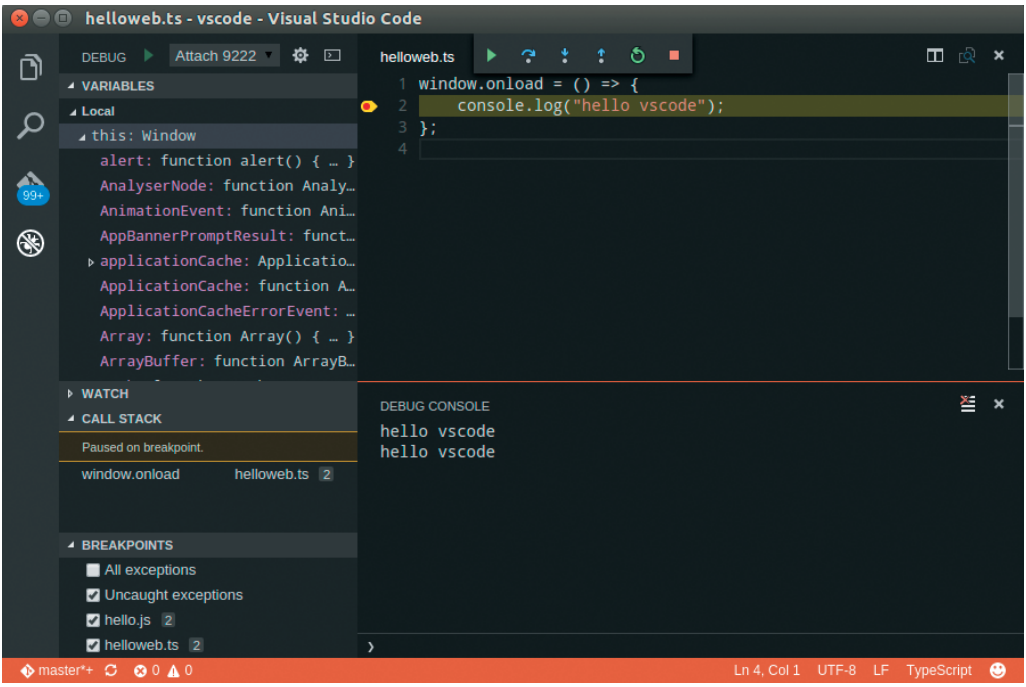
Следующим шагом является запуск Chrome с использованием параметра порта отладки. В системах Linux это делается из командной строки следующим образом:

```
google-chrome --remote-debugging-port=9222
```

Обратите внимание, что вам нужно убедиться, что другие экземпляры Chrome не запущены, чтобы использовать его в качестве отладчика с VSCode.

Затем загрузите файл `index.html` в браузере, используя комбинацию клавиш *Ctrl+O*, и выберите файл для загрузки. Вы должны увидеть файл HTML, отображающий текст `hello vscode`.

Теперь мы можем вернуться к VSCode, щелкнуть по значку отладки и выбрать опцию **Attach Chrome** в раскрывающемся списке средства запуска. Нажмите клавишу *F5*, и теперь отладчик VSCode должен быть подключен к работающему экземпляру Chrome. Затем нам нужно будет обновить страницу в Chrome, чтобы начать отладку:



Немного подправив наш файл launch.json, мы можем объединить эти действия, выполненные вручную, в единое средство запуска следующим образом:

```
{
   "name": "Launch chrome",
   "type": "chrome",
   "request" : "launch",
   "url" : "file:///...insert full path here.../index.html",
   "runtimeArgs": [
      "--new-window",
      "--remote-debugging-port=9222"
   ],
   "sourceMaps": true
}
```

В данной конфигурации запуска мы изменили свойство request с `attach` на `launch`, которое запустит новый экземпляр Chrome и автоматически перейдет к пути к файлу, указанному в свойстве `url`. Свойство `runtimeArgs` теперь также определяет порт удаленной отладки 9222. Имея этот модуль запуска, мы можем просто нажать клавишу *F5*, чтобы запустить Chrome, с правильным URL-адресом и опциями отладки для отладки HTML-приложений.

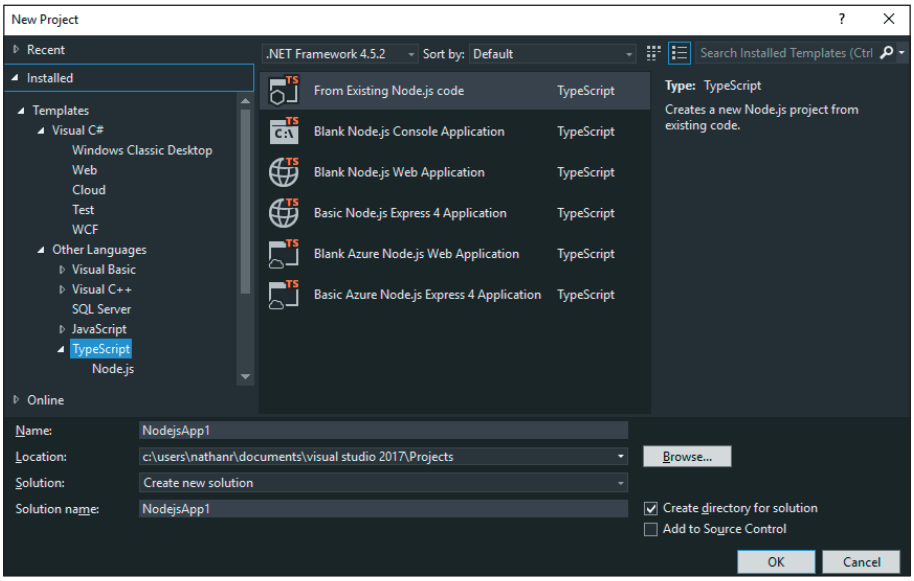
Microsoft Visual Studio

Давайте теперь посмотрим на Microsoft Visual Studio. Это основная интегрированная среда разработки от компании Microsoft, которая поставляется в различных ценовых комбинациях. На момент написания данной главы последняя версия Microsoft – Visual Studio 2017. У Microsoft есть модель лицензирования на основе Azure, от 45 долларов в месяц, вплоть до профессиональной лицензии с подпиской MSDN в размере около 1199 долларов. Хорошая новость состоит в том, что у Microsoft также есть версия Community Edition, которую можно использовать в некорпоративных средах как для бесплатных, так и для неоплачиваемых продуктов. Компилятор TypeScrip входит в состав всех этих версий.

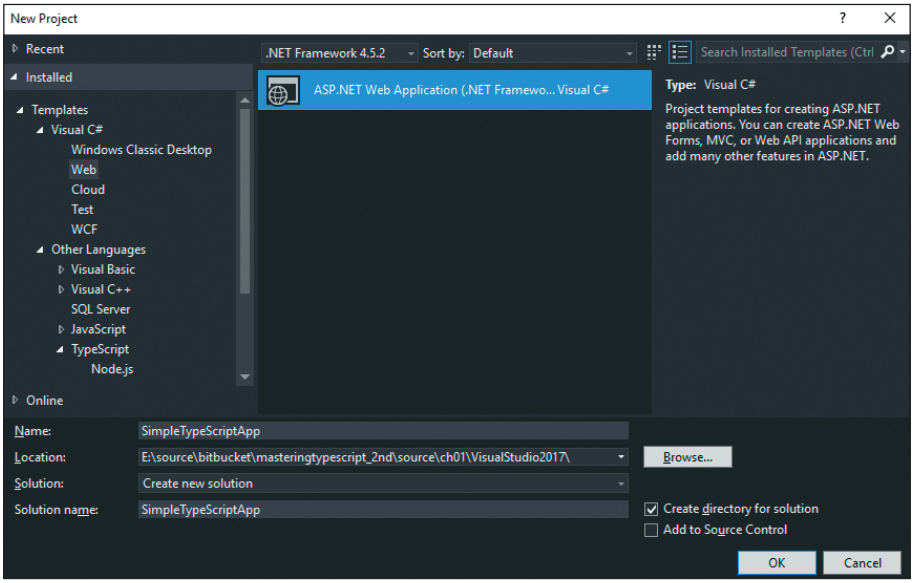
Visual Studio можно загрузить как веб-установщик или как образ компакт-диска `.iso`. Обратите внимание, что во время установки веб-установщику потребуется подключение к интернету, поскольку он загружает необходимые пакеты на этапе установки. Visual Studio также потребуется Internet Explorer 10 или более поздней версии, но во время установки вам будет предложено обновить версию браузера, если вы еще этого не сделали. Если вы используете установщик `.iso`, имейте в виду, что вам может потребоваться загрузить и установить дополнительные исправления для операционной системы, в случае если вы не обновляли ее в течение некоторого времени.

Создание проекта в Visual Studio

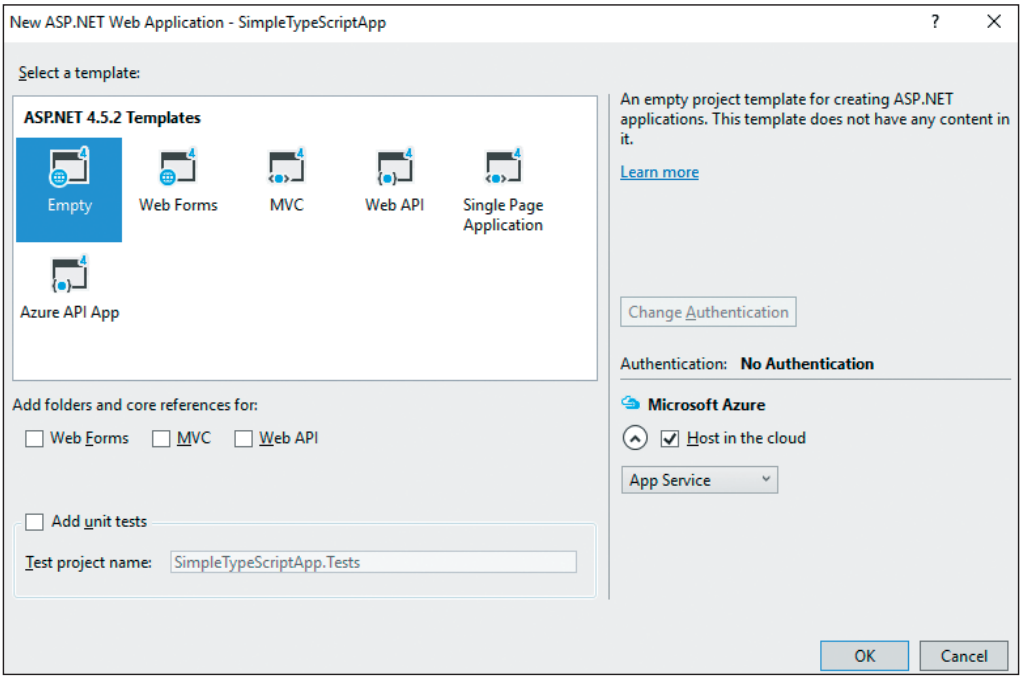
После установки Visual Studio 2017 запустите его и создайте новый проект (**File | New Project** (**Файл | Новый проект**)). Есть много различных опций, доступных для шаблонов новых проектов, в зависимости от выбора языка. В разделе **Templates** (**Шаблоны**) слева вы увидите параметр **Other Languages** (**Другие языки**), а под ним – **TypeScript**. Доступные шаблоны проекта в Visual Studio 2017 немного отличаются от тех, что были в Visual Studio 2015, и направлены на разработку Node. В состав Visual Studio 2015 входил шаблон **Html Application with TypeScript**, который создавал очень простое одностраничное веб-приложение. К сожалению, в Visual Studio 2017 эта опция была удалена:



Чтобы создать простое веб-приложение TypeScript в Visual Studio 2017, сначала нам нужно будет создать пустое веб-приложение, а затем при необходимости добавить файлы TypeScript в этот проект. В нашем диалоговом окне **Шаблоны** выберите параметр шаблона **Visual C#**, а затем выберите параметр **Web** (**Сеть**). Это даст нам шаблон проекта с именем **ASP.NET Web Application**. Выберите **имя** (**Name**) и **местоположение** (**Location**) для нового проекта, а затем нажмите **OK**, как показано на приведенном ниже скриншоте:



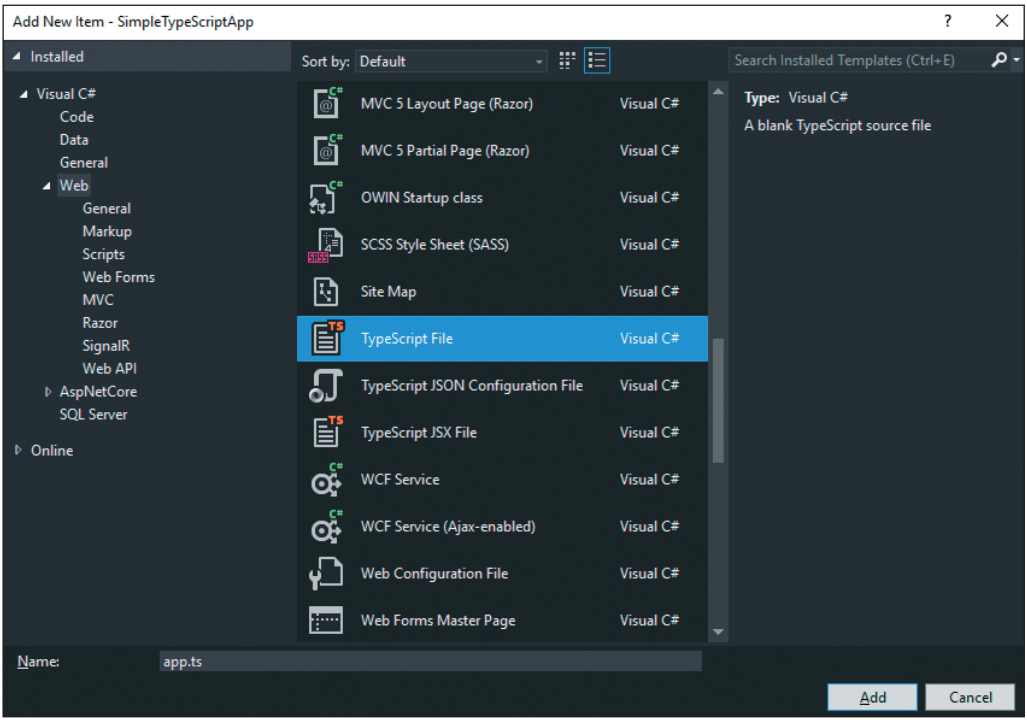
Как только мы выберем основную информацию для своего нового проекта, Visual Studio сгенерирует второе диалоговое окно, спрашивающее, какого рода проект **ASP.NET** мы хотели бы сгенерировать. Выберите пустой шаблон и нажмите **Ok**:



После этого в Visual Studio 2017 появится другое диалоговое окно **Create App Service** (**Создать службу приложения**), в котором представлены параметры для создания узла в Azure для вашего нового веб-приложения. Мы не будем публиковать наше приложение в Azure, поэтому на данном этапе можно нажать кнопку **Skip** (**Пропустить**).

Настройки проекта по умолчанию

После создания нового пустого веб-приложения ASP.NET мы можем приступить к добавлению файлов в проект, щелкнув правой кнопкой мыши сам проект и выбрав **Add** (**Добавить**), затем **New Element** (**Новый элемент**). Есть два файла, которые мы собираемся добавить в проект, а именно файл `index.html` и файл TypeScript `app.ts`. Для каждого из этих файлов выберите соответствующий шаблон Visual Studio, как показано ниже:



Теперь мы можем открыть файл `app.ts` и начать вводить следующий код:

```
class MyClass {
   public render(divId: string, text: string) {
      var el: HTMLElement = document.getElementById(divId);
      el.innerText = text;
   }
}

window.onload = () => {
   var myClass = new MyClass();
   myClass.render("content", "Hello World");
};
```

Здесь мы создали класс с именем `MyClass`, у которого есть одна функция `render`. Эта функция принимает два параметра с именами `divId` и `text`. Функция находит HTML-элемент DOM, соответствующий аргументу `divId`, а затем устанавливает для свойства `innerText` значение аргумента text. Потом мы определяем функцию, которая будет вызываться, когда браузер вызывает window.onload. Эта функция создает новый экземпляр класса `MyClass` и вызывает функцию render.

Не пугайтесь, если этот синтаксис и код немного сбивают с толку. Мы рассмотрим все языковые элементы и синтаксис в последующих главах. Цель данного упражнения – простое использование Visual Studio в качестве среды разработки для редактирования кода TypeScript.

Вы заметите, что Visual Studio имеет очень мощные опции Intellisense и будет предлагать код, имена функций или имена переменных по мере ввода кода. Если они не появляются автоматически, то после нажатия комбинации *Ctrl+пробел* появятся варианты Intellisense для кода, который вы в настоящее время набираете.

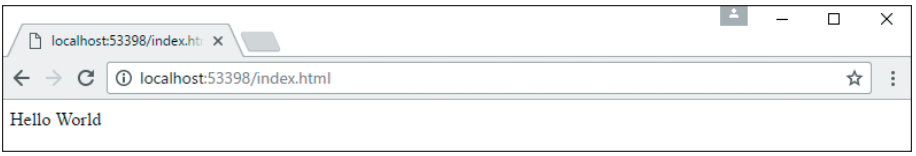
Имея в наличии файл `app.ts`, мы можем скомпилировать его, воспользовавшись сочетанием клавиш *Ctrl+Shift+B* или клавишей *F6*, либо выбрав опцию **Build** (**Сборка**) на панели инструментов. Если в коде TypesScript, который мы компилируем, есть какие-либо ошибки, Visual Studio автоматически откроет панель **Error List** (**Список ошибок**), показывающую текущие ошибки компиляции. После двойного щелчка по любой из этих ошибок на панели редактора откроется файл, и курсор автоматически переместится на код, вызывающий сбой.

Сгенерированный файл `app.js` не включен в Solution Explorer в Visual Studio. Включен только файл `app.ts`. Так задумано. Если вы хотите увидеть созданный файл JavaScript, просто нажмите кнопку **Show All Files** (**Показать все файлы**) на панели инструментов Solution Explorer.

Чтобы включить наш файл TypeScript на страницу HTML, нам нужно отредактировать файл `index.html` и добавить тег `<script>` для загрузки `app.js`:

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="utf-8" />
      <title></title>
      <script src="app.js"></script>
   </head>
   <body>
      <div id="content"></div>
   </body>
</html>
```

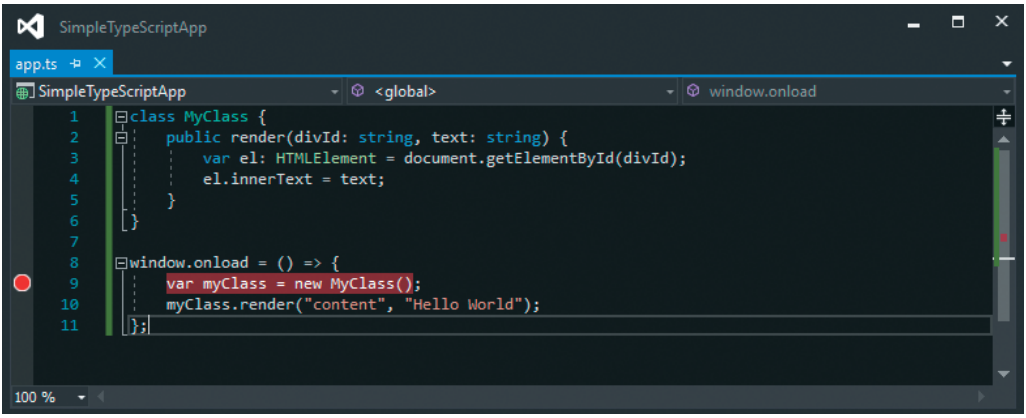
Здесь мы добавили тег `<script>` для загрузки своего файла `app.js`, а также создали элемент `<div>` с идентификатором `content`. Это элемент DOM, в котором наш код будет изменять свойство `innerHtml`. Теперь мы можем нажать клавишу *F5*, чтобы запустить наше приложение:



Отладка в Visual Studio

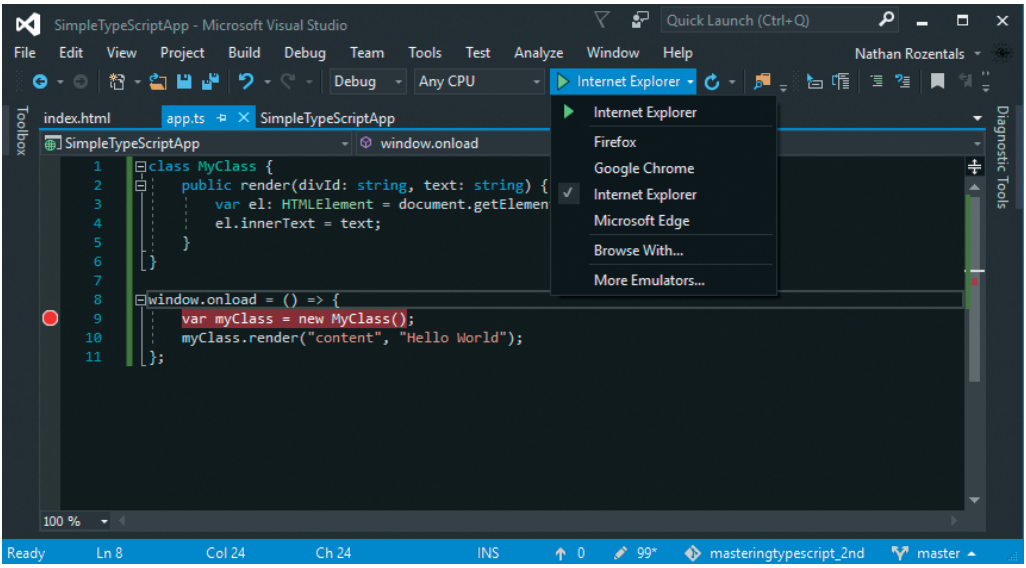
Одна из лучших функций Visual Studio состоит в том, что это – по-настоящему интегрированная среда. Отладка TypeScript в Visual Studio точно такая же, как отладка C# или любого другого языка в Visual Studio, и включает в себя обычные окна **Immediate**, **Locals**, **Watch** и **Call stack**.

Для отладки TypeScript в Visual Studio просто поместите точку останова на строке, которую вы хотите остановить в файле TypeScript (переместите указатель мыши в область точек останова рядом со строкой исходного кода и щелкните мышью). На приведенном далее скриншоте мы поместили точку останова в функцию `window.onload`. Чтобы начать отладку, просто нажмите клавишу *F5*:



Когда строка исходного кода выделена желтым цветом, просто наведите указатель мыши на любую из переменных в исходном коде или используйте окна Immediate, Watch, Locals или Call stack.

Обратите внимание, что Visual Studio поддерживает отладку только в Internet Explorer 11. Если на вашем компьютере установлено несколько браузеров (включая Microsoft Edge), убедитесь, что вы выбрали Internet Explorer в своей панели инструментов отладки, как показано на скриншоте ниже:



WebStorm

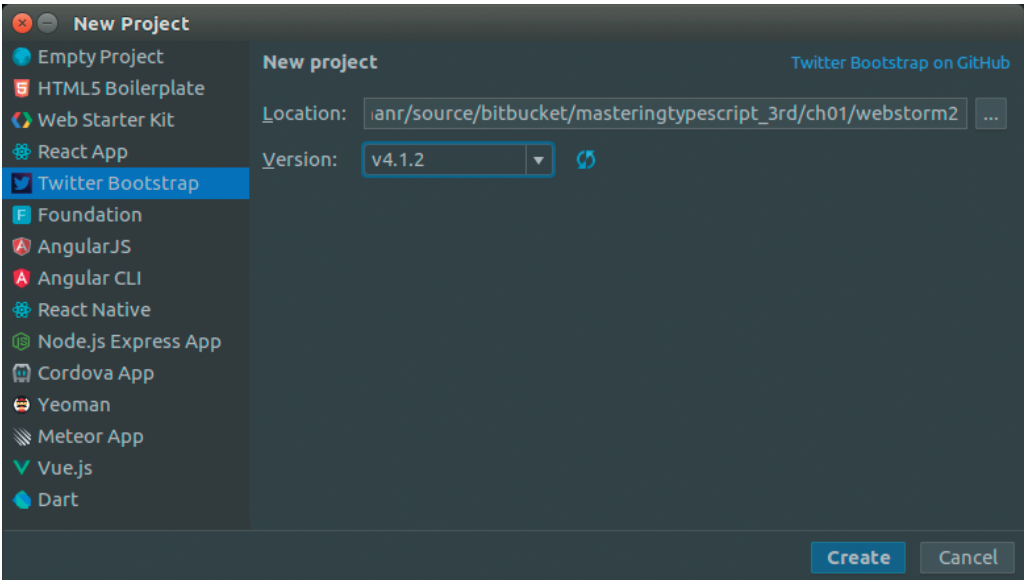
WebStorm – популярная интегрированная среда разработки от компании JetBrains (`http://www.jetbrains.com/webstorm`), которая будет работать в Windows, macOS и Linux. Цены варьируются от 59 долларов в год для одного разработчика до 129 долларов в год при приобретении коммерческой лицензии. JetBrains также предлагает 30-дневную пробную версию.

В WebStorm есть несколько замечательных функций, включая редактирование в режиме реального времени, рефакторинг кода и Intellisense. В частности, редактирование в режиме реального времени позволяет держать браузер открытым, который будет автоматически обновляться на основе изменений в CSS, HTML и JavaScript по мере ввода. Всплывающие подсказки, которые также доступны с другим популярным продуктом от JetBrains, ReSharper, выделят написанный вами код и предложат лучшие способы его реализации. WebStorm также имеет большое количество шаблонов проектов, которые легко интегрируются в среду разработки, автоматически загружая и включая соответствующие файлы JavaScript или CSS в ваш проект.

В системах Windows настроить WebStorm так же просто, как загрузить пакет с веб-сайта и запустить установщик. В системах Linux Webstorm предоставляется в виде tar-архива. После распаковки установите WebStorm, запустив скрипт `webstorm.sh` в каталоге `bin`. Обратите внимание, что в системах Linux перед продолжением установки должна быть установлена работающая версия Java.

Создание проекта в WebStorm

Чтобы создать проект WebStorm, запустите WebStorm и нажмите **File | New Project** (**Файл | Новый проект**). Выберите шаблон в левом меню и заполните параметры конфигурации на панели справа. В зависимости от того, какой шаблон выбран для проекта, параметры конфигурации изменятся. Для данного проекта мы выберем шаблон **Twitter Bootstrap**, для которого требуется только местоположение и версия bootstrap:



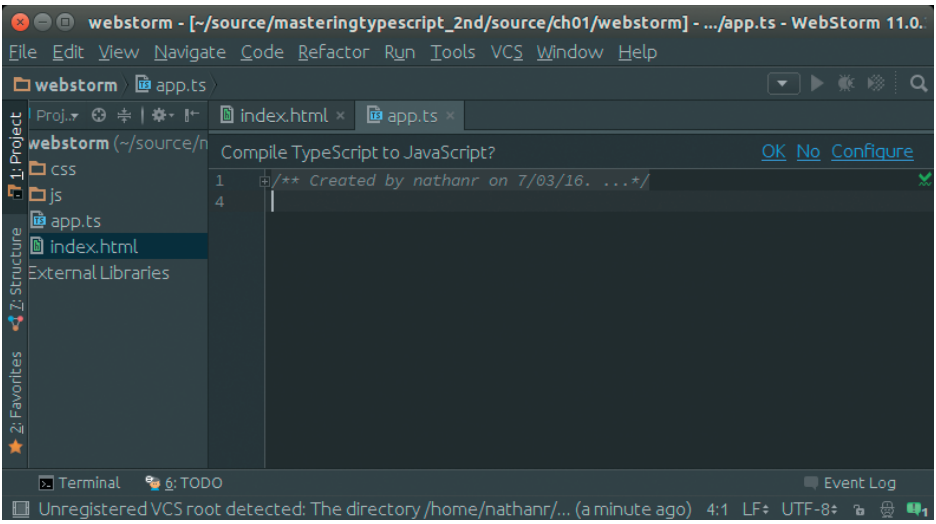
Файлы по умолчанию

После выбора шаблона проекта WebStorm создаст дерево каталогов проекта по умолчанию и загрузит необходимые файлы, чтобы мы могли начать использовать этот шаблон. В случае с проектом Bootstrap обратите внимание, что Webstorm удобно создал каталог `css` и `js` и автоматически загрузил и включил соответствующие файлы CSS и JavaScript. Обратите внимание, что шаблон проекта не создал для нас файл `index.html` и не создал никаких файлов TypeScript. Итак, давайте создадим файл `index.html`.

Просто нажмите на **File | New** (**Файл | Новый**), выберите HTML-файл, введите `index` в качестве имени и нажмите `OK`.

Далее давайте создадим файл TypeScript аналогичным образом. Выберите **Файл | Новый**, а затем файл TypeScript. Мы назовем этот файл `app` (или `app.ts`) для дублирования проекта Visual Studio, который мы создали ранее. Когда мы щелкнем внутри нового файла `app.ts`, WebStorm выведет сообщение в верхней ча-

сти файла с предложением **Compile TypeScript to JavaScript?** (**Компилировать TypeScript в JavaScript?**) с тремя вариантами – **OK**, **No** и **Configure**, как показано на скриншоте ниже:



При нажатии на **Configure** откроется панель **Settings** (**Настройки**) для TypeScript. На этой панели есть несколько опций, но на данный момент мы можем просто принять значения по умолчанию и нажать **Ok**.

Создание простого HTML-приложения

Теперь, когда мы настроили WebStorm для компиляции своих файлов Typescript, давайте создадим простой класс TypeScript и используем его для изменения свойства `innerText` HTML-элемента `div`. Во время набора кода вы заметите, среди прочего, функцию автозаполнения WebStorm или функцию Intellisense, которая помогает вам доступными ключевыми словами, параметрами и соглашениями об именовании. Это одна из самых мощных функций WebStorm, и она напоминает расширенную Intellisense, которую можно встретить в Visual Studio. Чтобы увидеть список ошибок компиляции TypeScript, мы можем открыть окно вывода TypeScript, перейдя в **View | Tool Windows | TypeScript** (**Вид | Окна инструментов | TypeScript**). Когда мы вводим код в этот файл, WebStorm автоматически скомпилирует наш файл (без необходимости его сохранения) и сообщит о любых ошибках в окне инструментов TypeScript. Идите дальше и введите приведенный ниже код TypeScript, в ходе чего вы получите прекрасное представление о доступном автозаполнении WebStorm и возможностях сообщения об ошибках:

```
class MyClass {
  public render(divId: string, text: string) {
```

```
        var el: HTMLElement | null = document.getElementById(divId);
        if (el) {
            el.innerText = text;
        }
    }
}


window.onload = () => {
    var myClass = new MyClass();
    myClass.render("content", "Hello World");
}
```

Этот код похож на пример, который мы использовали для Visual Studio 2017.

Если у вас есть какие-либо ошибки в вашем файле TypeScript, они автоматически отобразятся в окне вывода, предоставляя вам мгновенную обратную связь во время ввода кода.

Создав этот файл, мы теперь можем включить его в наш файл `index.html` и попробовать выполнить отладку.

Откройте файл `index.html` и добавьте тег `script`, чтобы включить файл `app.js`, а также `div` с идентификатором `content`. Как и в случае с редактированием TypeScript, вы обнаружите, что WebStorm обладает мощными функциями Intellisense и при редактировании HTML:

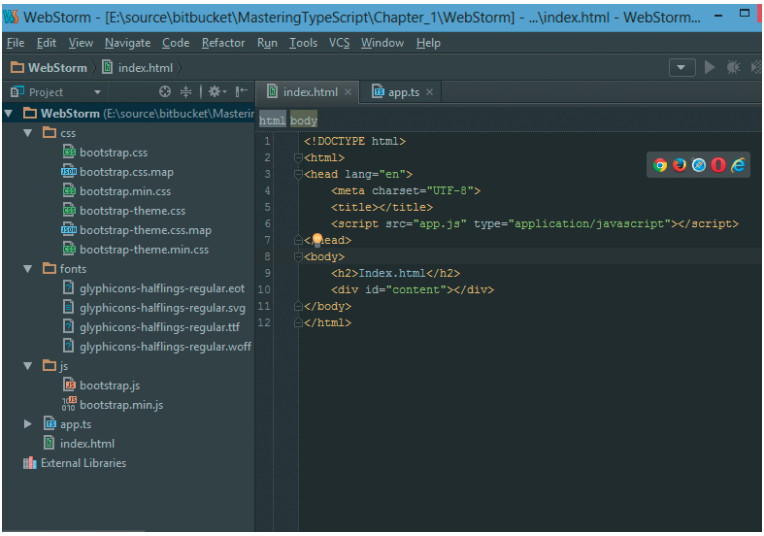
```
<!DOCTYPE html>
<html>
    <head lang="en">
        <meta charset="UTF-8">
        <title></title>
        <script src="app.js"></script>
    </head>
    <body>
        <div id="content"></div>
    </body>
</html>
```

Опять же, этот HTML-код такой же, как мы использовали ранее в примере с Visual Studio.

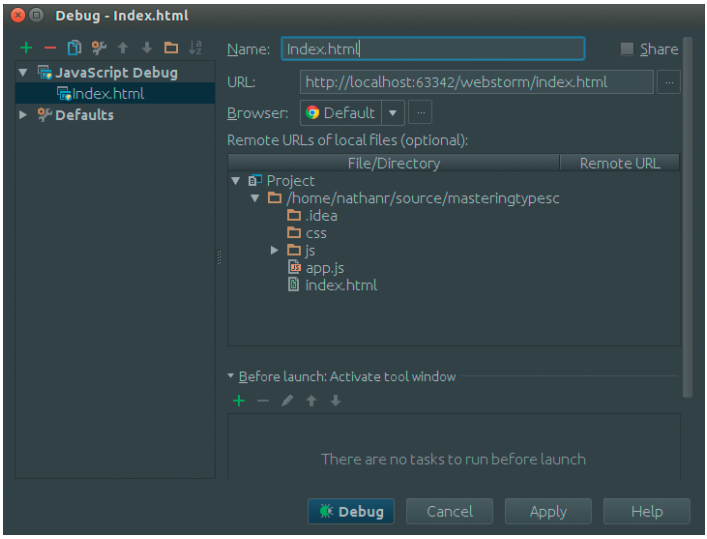
Запуск веб-страницы в Chrome

При просмотре или редактировании HTML-файлов в WebStorm вы увидите небольшой набор значков браузера, которые появляются в правом верхнем углу

окна редактирования. После нажатия на любую из иконок запустится ваша текущая HTML-страница, используя выбранный браузер:



Для отладки нашего веб-приложения в WebStorm нам потребуется настроить конфигурацию отладки для файла `index.html`. Нажмите на **Run | Debug** (**Запуск | Отладка**), а затем отредактируйте настройки. Нажмите кнопку со знаком «плюс» (+), выберите опцию **JavaScript debug** слева и дайте этой конфигурации название. Обратите внимание, что WebStorm уже определил, что `index.html` является страницей по умолчанию, но это легко изменить. Затем нажмите **Debug** (**Отладка**) в нижней части экрана, как показано на приведенном ниже скриншоте:



WebStorm использует плагин Chrome для включения отладки в Chrome и предложит вам в первый раз начать отладку, чтобы загрузить и включить плагин `JetBrains IDE Support`. Когда этот плагин включен, WebStorm располагает очень мощным набором инструментов для проверки кода JavaScript, добавления наблюдателей, просмотра консоли и многого другого непосредственно в среде разработки, как показано на приведенном ниже скриншоте:

Другие редакторы

Существует ряд редакторов, которые включают в себя поддержку TypeScript, такие как Atom, Brackets и даже старый редактор Vim. Каждый из этих редакторов имеет различные уровни поддержки TypeScript, в том числе подсветку синтаксиса и Intellisense. Использование этих редакторов представляет собой базовую среду разработки TypeScript, опирающуюся на использование командной строки для автоматизации задач сборки. Они не имеют встроенных средств отладки и, следовательно, не могут считаться **интегрированной средой разработки (IDE)** сами по себе, но могут легко использоваться для создания приложений TypeScript. Основной рабочий процесс с использованием этих редакторов будет следующим:

1. Создание и изменение файлов с использованием редактора.
2. Запуск компилятора TypeScript из командной строки.
3. Запуск или отладка приложений с использованием существующих отладчиков.

Использование --watch и Grunt

В чистой среде любое изменение в файле TypeScript означает, что нам нужно повторно вводить команду tsc из командной строки каждый раз, когда мы хотим скомпилировать наш проект.

Очевидно, что будет очень утомительно переключаться в командную строку и вручную компилировать наш проект всякий раз, когда мы вносим изменения. К счастью, компилятор TypeScript предоставляет опцию `--watch`, которая будет запускать фоновую задачу, чтобы отслеживать файлы на предмет изменений и автоматически перекомпилировать их при обнаружении изменений. Из командной строки мы можем запустить:

```
tsc --watch
```

Здесь мы вызвали компилятор TypeScript с помощью опции `--watch`, который затем запустит шаг компиляции в режиме просмотра. В тех случаях, когда мы изменяем `.ts`-файлы в нашем каталоге проекта, шаг компиляции будет выполнен повторно и сообщит о найденных ошибках.

Если у нас есть более сложный процесс сборки, который, например, компилирует наш проект, а затем необходимо удалить полученный JavaScript, мы можем использовать менеджер задач для автоматического выполнения рутинных операций, такой как Grunt или Gulp. В качестве примера этого процесса давайте скопируем опцию `--watch`, используя Grunt для автоматического вызова компилятора `tsc` при сохранении файла. Gulp очень похож на Grunt и в некоторых случаях может выполнять шаги быстрее, чем Grunt. Grunt, однако, обладает более простым синтаксисом для настройки, и поэтому мы будем использовать его в этом разделе, чтобы представить концепцию менеджеров задач для автоматического выполнения рутинных операций.

Grunt работает в среде Node и, следовательно, должен быть установлен как npm-зависимость нашего проекта. Чтобы установить Grunt, нам сначала нужно создать файл `packages.json` в базовом каталоге проекта, в котором будут перечислены все зависимости пакета npm, которые могут нам понадобиться. Чтобы создать файл `packages.json`, откройте командную строку, перейдите в базовый каталог своего проекта, а затем просто введите:

```
npm init
```

Далее следуйте инструкциям. Вы можете в значительной степени оставить все параметры в качестве значений по умолчанию и всегда возвращаться, чтобы отредактировать файл `packages.json`, созданный на этом шаге, если вам потребуется настроить какие-либо изменения.

Теперь, когда у нас создан файл `packages.json`, мы можем установить Grunt. В Grunt есть два компонента, которые должны быть установлены независимо. Во-первых, нам нужно установить интерфейс командной строки Grunt, который позволяет запускать Grunt из командной строки. Это может быть сделано следующим образом:

```
npm install -g grunt-cli
```

Второй компонент – установить файлы Grunt в каталог нашего проекта:

```
npm install grunt --save-dev
```

Опция `--save-dev` установит локальную версию Grunt в каталог проекта. Это сделано для того, чтобы несколько проектов на вашем компьютере могли использовать разные версии Grunt. Нам также понадобится установить пакет `grunt-exec` и пакет `grunt-contrib-watch`. Они могут быть установлены с помощью следующих команд:

```
npm install grunt-exec --save-dev
npm install grunt-contrib-watch --save-dev
```

Наконец, нам понадобится файл GruntFile.js. Используя редактор, создайте новый файл, сохраните его как GruntFile.js и введите приведенный ниже код. Обратите

внимание, что мы создаем здесь файл JavaScript, а не файл TypeScript. Вы можете найти копию этого файла в примере исходного кода, к этой главе:

```
module.exports = function (grunt) {
   grunt.loadNpmTasks('grunt-contrib-watch');
   grunt.loadNpmTasks('grunt-exec');
   grunt.initConfig( {
      pkg: grunt.file.readJSON('package.json'),
      watch : {
         files : ['**/*.ts'],
         tasks : ['exec:run_tsc']
      },
      exec: {
         run_tsc: {cmd:'tsc'}
      }
   });
   grunt.registerTask('default', ['watch']);
};
```

Файл `GruntFile.js` содержит простую функцию для инициализации среды Grunt и указания команд для запуска. Первые две строки функции загружают `gruntcontrib-watch` и `grunt-exec` как задачи npm. Затем мы вызываем `initConfig` для настройки запускаемых задач. В этом разделе конфигурации есть свойство `pkg`, свойство `watch` и свойство `exec`. Свойство `pkg` используется для загрузки файла `package.json`, который мы создали ранее в рамках шага `npm init`.

У свойства watch есть два подчиненных свойства. Свойство `files` определяет соответствующий алгоритм для Grunt, чтобы определить, какие файлы нужно искать. В этом случае он настроен на поиск любых файлов `.ts` в пределах всего нашего исходного дерева. Массив `tasks` указывает, что мы должны запустить команду `exec: run_tsc` после изменения файла. Наконец, мы вызываем `grunt.registerTask`, указывая, что задачей по умолчанию является отслеживание изменений файла.

Теперь мы можем запустить grunt из командной строки:

```
grunt
```

Как видно из вывода командной строки, Grunt выполняет задачу `watch` и ожидает изменений в любых файлах `.ts`:

```
Running "watch" task
Waiting...
```

Откройте любой файл TypeScript, внесите небольшое изменение (добавьте пробел или что-либо еще), а затем нажмите сочетание клавиш *Ctrl+S*, чтобы сохра-

нить файл. Теперь вернемся к выводу из командной строки Grunt. Вы должны увидеть что-то вроде этого:

```
>> File "hellogrunt.ts" changed.
Running "exec:run_tsc" (exec) task
Done, without errors.
Completed in 1.866s at Fri Jul 20 2018 22:22:52 GMT+0800 (AWST) -
Waiting...
```

Эти выходные данные командной строки являются подтверждением того, что задача наблюдения Grunt обнаружила, что файл `hellogrunt.ts` изменился, запустила задачу `exec:run_tsc` и ожидает изменения следующего файла. Теперь мы также должны увидеть файл `hellogrunt.js` в том же каталоге, что и наш файл Typescript.

Резюме

В этой главе мы кратко рассмотрели, что такое TypeScript и какие преимущества он может принести в процесс разработки JavaScript. Мы также рассмотрели настройку среды разработки с использованием ряда популярных интегрированных сред разработки и посмотрели, как будет выглядеть чистая среда разработки. Теперь, когда мы настроили среду разработки, мы можем начать рассматривать сам язык TypeScript более подробно. В следующей главе мы обсудим основные типы, доступные в TypeScript, а затем обсудим различные способы их использования с переменными или функциями.

Глава 2

Типы, переменные и методы функций

TypeScript использует сильную типизацию в JavaScript с помощью простого синтаксиса, который *Андерс Хейлсберг* называет синтаксическим сахаром. Сахар – это то, что назначает тип переменной. Синтаксис сильной типизации, который официально называется аннотацией типа, используется везде, где применяется переменная. Другими словами, мы можем использовать аннотацию типа в объявлении переменной, параметре функции или для описания типа возвращаемого значения самой функции.

Как мы обсуждали в `главе 1` *«Инструменты TypeScript и параметры фреймворков»*, принудительное применение типов в языке разработки дает множество преимуществ. К ним относятся улучшенная проверка ошибок, возможность интегрированной среды разработки предоставлять улучшенные подсказки и возможность вводить методы ООП в опыт кодирования.

Язык TypeScript использует несколько основных типов, таких как число и логический тип данных, а также несколько общих правил для определения типа переменной. Понимание этих правил и применение их в своем коде является фундаментальным навыком при написании кода TypeScript.

Наряду с этими основными правилами компилятор TypeScript также принимает синтаксис ES6 для более сложных манипулирований объектами. В этой главе представлены эти основные типы, правила, которые компилятор использует для проверки типов, а затем рассматриваются сложные манипулирования объектами.

Здесь мы рассмотрим следующие темы:

- основные типы и синтаксис типов – строки, числа и логические типы данных;
- типизация с поддержкой вывода типов и утиная типизация;
- шаблонные строки;
- массивы и использование `for ... in` и `for ... of`;
- тип any и явное приведение типов;
- перечисления;
- ключевые слова `const` и `let`;
- определенное присваивание и типы свойств с точечной нотацией;
- функции и анонимные функции;
- параметры функций;

- функции обратного вызова, сигнатуры функций и переопределения функций;
- конструкция try...catch;
- расширенные типы, включая объединенные типы, охранников типов и псевдонимы типов;
- типы never и unknown;
- Object rest and spread;
- кортежи;
- Bigint.

Если у вас уже есть опыт работы с TypeScript и хорошее понимание языка, то можете быстро прочитать всю главу. В противном случае вы можете перейти к концу главы, где рассматриваются более расширенные типы, такие как never.

Базовые типы

JavaScript по своей природе является динамически типизированным языком. Это означает, что любая конкретная переменная может содержать несколько типов данных, включая числа, строки, массивы, объекты и функции. Тип переменной в JavaScript определяется присваиванием. Это означает, что когда мы присваиваем значение переменной, интерпретатор JavaScript определяет тип этой переменной.

Однако среда выполнения JavaScript также может переназначать тип переменной в зависимости от того, как она используется или как она взаимодействует с другими переменными. Например, в некоторых случаях она может назначать номер строке.

Давайте посмотрим на пример этой динамической типизации в JavaScript и возникающие ошибки. Затем мы рассмотрим сильную типизацию, которую использует TypeScript, и его систему базовых типов.

Типизация в JavaScript

Как мы видели в `главе 1 «Инструменты TypeScript и параметры фреймворков»`, объекты и переменные JavaScript могут быть изменены или присвоены повторно на лету. В качестве примера рассмотрим приведенный ниже код:

```
function doCalculation(a,b,c) {
   return (a * b) + c;
}
var result = doCalculation(2,3,1);
console.log('doCalculation():' + result);
```

Здесь у нас есть функция `doCalculation`, которая вычисляет произведение аргументов `a` и `b`, а затем добавляет значение `c`. После мы вызываем функцию `doCalculation` с аргументами 2, 3 и 1 и записываем результат в консоль. Результат этого примера будет следующим:

```
doCalculation():7
```

Это ожидаемый результат: *2 * 3 = 6* и *6 + 1 = 7*. Теперь давайте посмотрим, что произойдет, если мы случайно вызовем функцию со строками вместо чисел:

```
result = doCalculation("2","3","1");
console.log('doCalculation():' + result);
```

Результат этого примера кода выглядит так:

```
doCalculation():61
```

Этот результат сильно отличается от ожидаемого нами результата, где должно быть 7. Так что же здесь происходит?

Если мы более пристально рассмотрим код в функции `doCalculation`, то начнем понимать, что делает JavaScript с нашими переменными и их типами.

Произведение двух чисел, то есть *(a * b)*, возвращает числовое значение; поэтому JavaScript автоматически преобразует значения 2 и 3 в числа для вычисления произведения и правильно вычисляет значение 6. Это особое правило, которое JavaScript применяет, когда нужно преобразовать строки в числа, и вступает в игру, когда результатом вычисления должно быть число. Однако символ сложения (+) можно использовать как с числами, так и со строками. Поскольку аргумент `c` является строкой, JavaScript преобразует значение 6 в строку, чтобы добавить две строки. Это приводит к добавлению строки 6 к строке 1, что дает 61.

Этот фрагмент кода является примером того, как JavaScript может изменять тип переменной в зависимости от того, как она используется. Это означает, что для эффективной работы с JavaScript нам необходимо знать о таком преобразовании типов и понимать, когда и где это преобразование может иметь место. Очевидно, что такого рода автоматическое преобразование типов может вызвать нежелательное поведение в нашем коде.

Типизация в TypeScript

TypeScript, с другой стороны, является сильно типизированным языком. Как только вы объявили переменную определенного типа, вы не можете ее изменить. Например, если вы объявляете переменную типа `string`, вы можете назначать ей только строковые значения, и любой экземпляр этой переменной может рассматривать ее только, как если бы она имела тип `string`. Это помогает убедиться

в том, что код, который мы пишем, будет работать так, как ожидается. Когда мы сразу узнаем, что значением переменной всегда будет строка, мы узнаем, какое из этих правил преобразования будет использовать JavaScript.

Программисты на JavaScript всегда полагались на документацию, чтобы понять, как вызывать функции, а также порядок и тип правильных параметров функции. Но что, если бы мы могли взять всю эту документацию и включить ее в интегрированную среду разработки?

Затем, когда мы пишем свой код, наш компилятор может автоматически указать нам, что мы неправильно используем переменные или функции в библиотеке JavaScript. Сделает ли это нас более эффективными, более производительными программистами и позволит нам генерировать качественный код с меньшим количеством ошибок?

TypeScript делает именно это. Он использует очень простой синтаксис для определения типа переменной, чтобы обеспечить ее правильное использование. Если мы нарушим любое из этих правил, компилятор TypeScript автоматически сгенерирует ошибки, указывая нам на строки в коде, где они находятся.

Вот как TypeScript получил свое имя. Это JavaScript с сильной типизацией, то есть TypeScript. Давайте посмотрим на этот очень простой синтаксис языка, который включает Type в TypeScript.

Синтаксис типов

Синтаксис TypeScript для объявления типа переменной должен включать двоеточие (`:`) после имени переменной, а затем указывать ее тип. В качестве примера давайте перепишем нашу проблемную функцию `doCalculation`, чтобы она принимала только числа. Рассмотрим приведенный ниже код:

```
function doCalculation(
   a : number,
   b : number,
   c : number) {
      return ( a * b ) + c;
   }
var result = doCalculation(3,2,1);
console.log("doCalculation():" + result);
```

Здесь мы указали, что функция `doCalculation` должна вызываться с тремя числами. Это означает, что свойства `a`, `b` и `c` должны иметь тип `number` для вызова этой функции. Если мы сейчас попытаемся вызвать эту функцию со строками, как мы это сделали в примере с кодом на JavaScript:

```
var result = doCalculation("1", "2", "3");
console.log("doCalculation():"" + result);
```

то компилятор TypeScript выдаст следующую ошибку:

```
error TS2345: Argument of type '"1"' is not assignable to
parameter of type 'number'.
```

Это сообщение об ошибке ясно говорит нам, что мы не можем передать строку в нашу функцию, где ожидается числовое значение.

Чтобы дополнительно проиллюстрировать этот момент, рассмотрим следующий код:

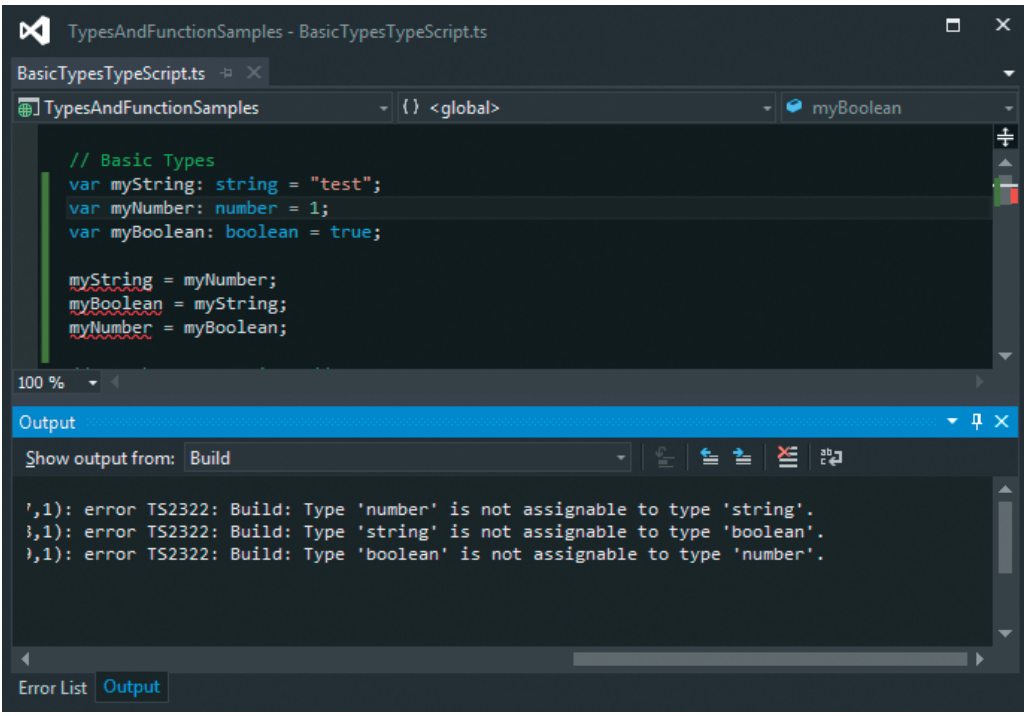
```
var myString : string;
var myNumber : number;
var myBoolean : boolean;
myString = "1";
myNumber = 1;
myBoolean = true;
```

Здесь мы сообщаем компилятору, что тип переменной `myString` – `string`, еще до того, как сама переменная была использована. Точно так же переменная `myNumber` имеет тип `number`, а переменная `myBoolean` – тип `boolean`. TypeScript ввел ключевые слова string, number и boolean для каждого из этих основных типов JavaScript.

Если затем мы попытаемся присвоить значение переменной другого типа, компилятор TypeScript выдаст ошибку времени компиляции. Учитывая переменные, объявленные в предыдущем коде, рассмотрим следующий код:

```
myString = myNumber;
myBoolean = myString;
myNumber = myBoolean;
```

Вот что происходит, если мы попытаемся смешать эти основные типы:



Компилятор TypeScript теперь начал генерировать ошибки компиляции, потому что обнаружил, что мы пытаемся смешать эти основные типы:

- первая ошибка генерируется, потому что мы не можем присвоить значение `number` переменной типа `string`;
- точно так же вторая ошибка компиляции означает, что мы не можем присвоить значение `string` переменной типа `boolean`;
- и третья ошибка генерируется, потому что мы не можем присвоить значение `boolean` переменной типа `number`.

Синтаксис сильной типизации, который использует язык TypeScript, означает, что мы должны убедиться, что типы в левой части оператора присваивания (=) совпадают с типами в правой части оператора.

Чтобы исправить предыдущий код и убрать ошибки компиляции, нам нужно сделать что-то вроде этого:

```
myString = myNumber.toString();
myBoolean = (myString === "test");
if (myBoolean) {
   myNumber = 1;
}
```

Здесь первая строка кода была изменена, чтобы вызвать функцию `.toString()` для переменной `myNumber` (тип `number`), дабы вернуть значение, имеющее тип `string`. Эта строка кода не выдает ошибку компиляции, поскольку обе стороны знака равенства (или оператора присваивания) являются строками.

Вторая строка кода также была изменена, так что правая часть оператора присваивания возвращает результат логического сравнения, `myString === "test"`, который вернет значение типа `boolean`. Следовательно, компилятор разрешит этот код, потому что обе стороны присваивания возвращают значение типа `boolean`.

Последняя строка нашего фрагмента кода была изменена, чтобы присвоить значение `1` (которое имеет тип number) переменной `myNumber`, если значение переменной `myBoolean` равно `true`.

Эту особенность *Андерс Хейлсберг* называет синтаксическим сахаром. Другими словами, с небольшим дополнением к сопоставимому коду JavaScript, TypeScript позволил нашему коду преобразоваться в сильно типизированный язык. Всякий раз, когда вы нарушаете эти строгие правила типизации, компилятор генерирует ошибки при наличии неправильного кода.

Типизация с поддержкой вывода типов

TypeScript также использует технику под названием типизация с поддержкой вывода типов для определения характера переменной. Это означает, что даже если мы не укажем тип, компилятор проверит, где переменная была впервые использована, а затем использует этот тип для оставшейся части блока кода. В качестве примера рассмотрим приведенный ниже код:

```
var inferredString = "this is a string";
var inferredNumber = 1;
inferredString = inferredNumber;
```

Мы начинаем с объявления переменной с именем `inferredString` и присваиваем ей строковое значение. TypeScript определяет, что этой переменной было присвоено значение типа `string`, и сделает вывод, что переменная `inferredString` имеет тип `string`. Следовательно, в ходе дальнейшего использования этой переменной необходимо будет рассматривать ее как строковый тип. У нашей второй переменной, `inferredNumber`, есть присвоенное ей число. Опять же, TypeScript выводит тип этой переменной как number.

Если затем мы попытаемся присвоить переменную `inferredString` (типа `string`) переменной `inferredNumber` (типа `number`) в последней строке кода, TypeScript выдаст знакомое сообщение об ошибке:

```
error TS2322: Type 'number' is not assignable to type 'string'
```

Данная ошибка генерируется вследствие правил типизации с поддержкой вывода типов TypeScript.

Утиная типизация

TypeScript также использует метод, называемый утиной типизацией, для более сложных типов переменных. Утиная типизация предполагает, что если это выглядит как утка и ходит как утка, то это, вероятно, и есть утка. Другими словами, две переменные считаются равными, если они имеют одинаковые свойства и методы, поскольку они могут быть взаимозаменяемыми, но все равно будут вести себя одинаково. Рассмотрим следующий код:

```
var complexType = { name: "myName", id: 1 };
complexType = { id: 2, name: "anotherName" };
```

Мы начнем с переменной `complexType`, которой был присвоен простой объект JavaScript с атрибутами `name` и `id`. Во второй строке кода мы переназначаем значение этой переменной `complexType` другому объекту, у которого также есть атрибуты `id` и `name`. В этом случае компилятор будет использовать утиную типизацию, чтобы выяснить, допустимо ли это назначение. Другими словами, если объект обладает тем же набором атрибутов, что и другой объект, то считается, что он относится к тому же типу.

Чтобы далее проиллюстрировать этот момент, давайте посмотрим, как реагирует компилятор, если мы пытаемся присвоить объект нашей переменной `complexType`, который не соответствует данному виду типизации:

```
var complexType = { name: "myName", id: 1 };
complexType = { id: 2 };
```

Здесь первая строка этого фрагмента кода определяет нашу переменную `complexType` и присваивает ей объект с атрибутами `id` и `name`. С этого момента TypeScript будет использовать этот выводимый тип для переменной `complexType`. Во второй строке кода мы пытаемся переназначить переменную `complexType` значению, у которого есть только атрибут `id`, но нет `name`. Эта строка кода выдает следующую ошибку компиляции:

```
error TS2322: Type '{ id: number; }' is not assignable to type
'{ name:string; id: number; }'.
Property 'name' is missing in type '{ id: number; }'.
```

Сообщение об ошибке довольно очевидно. В этом случае TypeScript использует утиную типизацию для обеспечения безопасности типов. Поскольку переменная `complexType` содержит как `id`, так и `name`, любой объект, который ей назначен, также должен содержать их.

Обратите внимание, что приведенный ниже код тоже выдает сообщение об ошибке:

```
var complexType = { name: "myName", id: 1 };
complexType = { name : "extraproperty", id : 2, extraProp: true };
```

Сгенерированная здесь ошибка выглядит так:

```
error TS2322: Type '{ name: string; id: number;
extraProp: boolean; }' is not assignable to type '{ name:
string; id: number; }'.

Object literal may only specify known properties, and 'extraProp'
does not exist in type '{ name: string; id: number; }'.
```

Как видно из этого сообщения, у переменной `complexType` нет свойства `extraProp`, и поэтому присваивание не выполняется.

Типизация с поддержкой вывода типов и утиная типизация – мощные возможности языка TypeScript, обеспечивающие сильную типизацию переменных, которые состоят из множества свойств, и все это без необходимости использовать явную типизацию.

Шаблонные строки

Прежде чем мы продолжим обсуждение типов, давайте рассмотрим простой метод записи значения любой переменной в консоль. Стоит отметить, что TypeScript допускает синтаксис строки шаблона ES6, который удобен для ввода значений в строки. Рассмотрим следующий код:

```
var myVariable = "test";
console.log("myVariable=" + myVariable);
```

Здесь мы просто присваиваем значение переменной, а затем записываем результат в консоль. Обратите внимание, что функция `console.log` использует немного форматирования, чтобы сделать сообщение читабельным, путем объединения строк с помощью синтаксиса "`string`" + `variable`. Давайте теперь посмотрим на эквивалентный код, который использует синтаксис строки шаблона ES6:

```
var myVariable = "test";
console.log('myVariable=${myVariable}');
```

Во второй строке этого фрагмента кода мы записываем некоторую информацию в консоль и используем синтаксис строки шаблона. Здесь следует отметить два важных момента. Во-первых, при определении строки мы использовали обратный штрих (`) вместо двойных кавычек ("). Использование обратного штриха дает сигнал компилятору TypeScript, что он должен искать значения шабло-

на в строке, заключенной в эти символы, и заменять их фактическими значениями. Во-вторых, мы использовали специальный `${...}` синтаксис в строке для обозначения шаблона. TypeScript вставит значение любой переменной, которая в данный момент находится в области видимости, в строку за нас. Это удобный метод работы с форматированием строк.

Компилятор TypeScript проанализирует стиль строковых шаблонов ES6 и сгенерирует код JavaScript, который использует стандартную конкатенацию строк. Таким образом, синтаксис строкового шаблона ES6 может использоваться независимо от целевой версии JavaScript.

В оставшейся части этой главы мы будем использовать синтаксис строкового шаблона всякий раз при записи значений в консоль.

Еще один удобный способ использования синтаксиса строки шаблона – это когда нам нужно записать весь сложный объект, включая все его свойства, в консоль. Рассмотрим приведенный ниже код:

```
var complexObject = {
   id: 2,
   name: 'testObject'
}
console.log('complexObject = ${JSON.stringify(complexObject)}');
```

Здесь мы определяем объект с именем `complexObject` и атрибутами `id` и `name`. Затем мы используем синтаксис строки шаблона для записи значения данного объекта в консоль. Но в этом примере мы вызываем функцию `JSON.stringify` и передаем переменную `complexObject` в качестве параметра. Это означает, что все, что находится внутри `${...}`, не ограничивается простыми типами, но может быть валидным кодом TypeScript. Вывод этого кода выглядит так:

```
complexObject = {"id":2,"name":"testObject"}
```

Здесь видно, что результатом вызова функции `JSON.stringify` является строковое представление всего объекта, которое мы записываем в консоль.

Массивы

Помимо основных типов `string`, `number` и `boolean` в JavaScript, в TypeScript также есть два других основных типа данных, которые мы сейчас рассмотрим подробнее, – массивы и перечисления. Давайте посмотрим на синтаксис для определения массивов.

Массив просто помечается нотацией `[]`, как и в JavaScript, и каждый массив может быть сильно типизирован для хранения значений определенного типа, как показано ниже:

```
var arrayOfNumbers: number [] = [1,2,3];
arrayOfNumbers = [3,4,5,6,7,8,9];
console.log(`arrayOfNumbers: ${arrayOfNumbers}`);
arrayOfNumbers = ["1", "2", "3"];
```

Здесь мы начинаем с определения массива с именем `arrayOfNumbers` и далее определяем, что каждый элемент этого массива должен иметь тип `number`. Затем мы переназначаем этот массив для хранения различных числовых значений. Интересно, что мы можем назначить массиву любое количество элементов, если каждый элемент имеет правильный тип. Потом мы используем простую строку шаблона для вывода значения массива в консоль.

Однако последняя строка в этом фрагменте выдаст следующее сообщение об ошибке:

```
error TS2322: Type 'string[]' is not assignable to type 'number[]'.
```

Данное сообщение предупреждает нас о том, что переменная `arrayOfNumbers` сильно типизирована, чтобы принимать только значения типа number. Поскольку наш код пытается присвоить массив строк этому массиву чисел, генерируется ошибка. Вывод этого фрагмента кода выглядит так:

```
arrayOfNumbers: 3,4,5,6,7,8,9
```

for...in и for...of

При работе с массивами обычной практикой является перебор элементов массива для выполнения задачи. Обычно оно выполняется внутри цикла `for` путем манипулирования индексом массива, как показано ниже:

```
var arrayOfStrings : string[] = ["first", "second", "third"];
for( var i = 0; i < arrayOfStrings.length; i++ ) {
   console.log(`arrayOfStrings[${i}] = ${arrayOfStrings[i]}`);
}
```

Здесь у нас есть массив с именем `arrayOfStrings` и стандартный цикл `for`, который использует переменную `i` в качестве индекса в нашем массиве. Мы получаем доступ к элементу массива, используя синтаксис `arrayOfStrings[i]`. Вывод этого кода выглядит так:

```
arrayOfStrings[0] = first
arrayOfStrings[1] = second
arrayOfStrings[2] = third
```

TypeScript также может использовать синтаксис ES6 `for ... in`, чтобы упростить перебор массивов. Вот пример цикла `for`, о котором шла речь выше, выраженного с использованием нового синтаксиса:

```
for( var itemKey in arrayOfStrings) {
   var itemValue = arrayOfStrings[itemKey];
   console.log(`arrayOfStrings[${itemKey}] = ${itemValue}`);
}
```

Здесь мы упростили цикл `for`, используя синтаксис `itemKey in arrayOfStrings`. Обратите внимание, что значение переменной `itemKey` будет перебирать ключи массива, а не сами элементы массива. В цикле `for` мы сначала разыменуем массив для извлечения значения массива для переменной `itemKey`, а затем записываем `itemKey` и `itemValue` в консоль. Вывод этого кода выглядит так:

```
arrayOfStrings[0] = first
arrayOfStrings[1] = second
arrayOfStrings[2] = third
```

Если нам не обязательно знать ключи массива и нас просто интересуют значения, содержащиеся в массиве, мы можем еще больше упростить перебор массивов, используя синтаксис `for ....of`. Рассмотрим следующий код:

```
for( var arrayItem of arrayOfStrings ) {
   console.log(`arrayItem = ${arrayItem}`);
}
```

Здесь мы используем синтаксис `for ...of` для итерации каждого значения массива `arrayOfStrings`. Каждый раз, когда выполняется цикл `for`, переменная `arrayItem` будет содержать следующий элемент в массиве. Вывод этого кода выглядит так:

```
arrayItem = first
arrayItem = second
arrayItem = third
```

Тип any

Вся эта проверка типов прекрасна и замечательна, но в JavaScript нет сильной типизации, и он позволяет смешивать и сопоставлять переменные. Рассмотрим приведенный ниже фрагмент кода, который на самом деле является валидным кодом JavaScript:

```
var item1 = { id: 1, name: "item 1" };
item1 = { id: 2 };
```

Здесь мы назначаем объект со свойством `id` и свойством `name` переменной `item1`. Затем мы переназначаем эту переменную объекту, у которого есть свойство `id`, но нет свойства `name`. К сожалению, как мы видели ранее, это невалидный код TypeScript, который приведет к следующей ошибке:

```
error TS2322: Type 'string[]' is not assignable to type 'number[]'.
```

Для таких случаев TypeScript использует тип `any`. Указание, что объект имеет тип `any`, по сути, ослабляет проверку на предмет сильной типизации со стороны компилятора. Следующий код показывает, как использовать тип any:

```
var item1 : any = { id: 1, name: "item 1" };
item1 = { id: 2 };
```

Обратите внимание, как изменилась первая строка кода. Мы указываем тип переменной `item1` как `: any`. Это специальное ключевое слово TypeScript позволяет переменной следовать свободно определенным правилам типов JavaScript, поэтому что угодно может быть присвоено чему угодно. Без спецификатора типа `: any` вторая строка кода обычно выдает ошибку.

Явное приведение типов

Как и для любого сильно типизированного языка, наступает момент, когда вам необходимо явно указать тип объекта. Эта концепция будет более подробно рассмотрена в следующей главе, но здесь стоит кратко остановиться на явном приведении типов. Объект может быть приведен к типу другого объекта с использованием синтаксиса `< >`.

Это не приведение в самом строгом смысле этого слова; это больше утверждение, которое используется во время компиляции компилятором TypeScript. Любое явное приведение, которое вы используете, будет скомпилировано в результирующем коде JavaScript и не повлияет на код во время выполнения.

Давайте перепишем наш предыдущий пример и используем явное приведение, как показано ниже:

```
var item1 = <any>{ id: 1, name: "item 1" };
item1 = { id: 2 };
```

Здесь мы заменили спецификатор типа `:any` в левой части присваивания явным приведением `<any>` в правой части. Это говорит компилятору явно рассматривать объект `{id: 1, name: "item 1"}` в правой части оператора присваивания как тип `any`. По сути, этот синтаксис эквивалентен нашим приведенным ранее примерам и определяет тип переменной `item1` как `any` (из-за правил типизации с поддержкой вывода типов). Затем это позволяет нам присвоить объект только со свойством `{id: 2}` переменной item1 во второй строке кода. Этот метод использования синтаксиса `< >` в правой части присваивания называется явным приведением.

В то время как тип any является необходимой функцией языка TypeScript и используется для обратной совместимости с JavaScript, его применение действи-

тельно должно быть максимально ограничено. Как мы видели в нетипизированном JavaScript, чрезмерное использование типа any быстро приведет к ошибкам кодирования, которые будет трудно обнаружить. Вместо того чтобы использовать тип any, попробуйте выяснить правильный тип объекта, который вы используете, а затем используйте этот тип.

В наших командах программистов мы используем аббревиатуру **Simply Find an Interface for the Any Type** (**S.F.I.A.T**). Произносится как «`sweat`». Это может показаться глупым, но дает понять, что тип any всегда следует заменять интерфейсом, поэтому просто найдите его. Интерфейс – это способ определения пользовательских типов в TypeScript, о которых речь пойдет в следующей главе. Просто помните, что, активно пытаясь определить, каким должен быть тип объекта, мы создаем сильно типизированный код и, следовательно, защищаем себя от ошибок кодирования в будущем.

Говоря кратко, избегайте тип any любыми способами.

Перечисления

Перечисления – это специальный тип, заимствованный из других языков, таких как C#, C ++ и Java, который обеспечивает решение проблемы специальных чисел. enum связывает удобочитаемое имя для определенного числа. Рассмотрим следующий код:

```
enum DoorState {
   Open,
   Closed,
   Ajar
}
```

Здесь мы определили перечисление с именем `DoorState` для представления состояния двери. Допустимые значения для этого состояния двери: `Open`, `Closed` или `Ajar`. Под капотом (в сгенерированном JavaScript) TypeScript назначит числовое значение каждому из этих понятных человеку значений перечисления. В этом примере значение перечисления `DoorState.Open` будет равно числовому значению `0`. Аналогично значение перечисления `DoorState.Closed` будет равно числовому значению `1`, а значение перечисления `DoorState.Ajar` будет равно `2`. Давайте быстро посмотрим, как мы будем использовать эти значения перечисления:

```
var openDoor = DoorState.Open;
console.log(`openDoor is: ${openDoor}`);
```

Здесь первая строка этого фрагмента кода создает переменную с именем `openDoor` и устанавливает ее значение в `DoorState.Open`. Вторая строка просто записывает значение переменной `openDoor` в консоль. Вывод будет выглядеть так:

```
openDoor is: 0
```

Это ясно показывает, что компилятор TypeScript заменил значение перечисления `DoorState.Open` на числовое значение `0`.

Теперь давайте используем это перечисление немного иначе:

```
var closedDoor = DoorState["Closed"];
console.log(`closedDoor is : ${closedDoor}`);
```

В этом фрагменте кода используется строковое значение `"Closed"` для поиска типа `enum` и присваивает результирующее значение `enum` переменной `closedDoor`. Вывод этого кода будет таким:

```
closedDoor is : 1
```

Этот пример ясно показывает, что значение enum для `DoorState.Closed` совпадает со значением `enum` для `DoorState["Closed"]`, поскольку в обоих случаях возвращается числовое значение `1`.

Перечисления – это удобный способ определения легко запоминающегося, удобочитаемого имени для специального числа. Использование удобочитаемых перечислений вместо простого разбрасывания различных специальных чисел в нашем коде делает цель кода более понятной. Использовать значение для всего приложения с именем `DoorState.Open` или `DoorState.Closed` гораздо проще, чем не забывать установить значение `0` для `Open`, `1` для `Closed` и `3` для `Ajar`. Помимо того что мы делаем наш код более читабельным и более понятным, использование перечислений также защищает нашу кодовую базу каждый раз, когда эти специальные числовые значения изменяются, поскольку все они определены в одном месте.

Последнее замечание касательно перечислений – это то, что мы можем установить числовое значение вручную, если это необходимо, следующим образом:

```
enum DoorState {
   Open = 3,
   Closed = 7,
   Ajar = 10
}
```

Здесь мы переопределили значения перечисления по умолчанию, чтобы установить значение `DoorState.Open` равным `3`, `DoorState.Closed` равным *7* и `DoorState.Ajar` – `10`.

Const enum

Облегченный вариант типа enum – это const enum, который просто добавляет ключевое слово const перед определением enum:

```
const enum DoorStateConst {
   Open,
   Closed,
   Ajar
}
var constDoorOpen = DoorStateConst.Open;
console.log(`constDoorOpen is : ${constDoorOpen}`);
```

`const enum` был введен в основном по соображениям производительности. Давайте быстро рассмотрим код JavaScript, сгенерированный из перечисления `DoorStateConst`:

```
var constDoorOpen = 0 /* Open */;
```

Обратите внимание, что компилятор просто вернул перечислению `DoorStateConst.Open` внутреннее значение `0` и полностью удалил определение `const enum`.

Строковые перечисления

Еще одним вариантом типа enum является строковое перечисление, при котором числовые значения заменяются строками:

```
enum DoorStateString {
   Open = "open",
   Closed = "closed",
   Ajar = "ajar"
}

var openDoorString = DoorStateString.Open;
console.log(`openDoorString = ${openDoorString}`);
```

Здесь у нас есть перечисление с именем `DoorStateString`, где каждое из значений перечисления теперь имеет тип `string`. Вывод этого фрагмента кода будет таким:

```
openDoorString = open
```

Как и ожидалось, компилятор TypeScript возвращает строку `"open"`.

Реализация перечислений

Когда TypeScript генерирует JavaScript для перечислений, он использует шаблоны замыканий и **немедленно вызываемых функций** для определения объекта JavaScript с правильными свойствами, которые можно использовать в качестве перечисления. Давайте посмотрим на эту реализацию. Учитывая наше перечисление `DoorState`, которое определяется следующим образом:

```
enum DoorState {
   Open,
   Closed,
   Ajar
}
```

мы знаем, что можем обратиться к значению `DoorState.Open`, которое будет представлено в виде числового значения `0`, `Closed` как `1` и `Ajar` как `2`. Но что произойдет, если мы будем обращаться к перечислению с использованием синтаксиса типа массива, как показано ниже:

```
var ajarDoor = DoorState[2];
console.log(`ajarDoor is : ${ajarDoor}`);
```

Здесь мы присваиваем переменной `ajarDoor` значение перечисления на основе второго значения индекса перечисления `DoorState`. Вывод этого кода, однако, вызывает удивление:

```
ajarDoor is : Ajar
```

Возможно, вы ожидали, что результат будет просто 2, но здесь мы получаем строку `Ajar`, которая является строковым представлением нашего исходного значения перечисления. Это на самом деле маленький изящный трюк, позволяющий нам получить доступ к строковому представлению вместо простого числа. Причина, по которой это возможно, заключается в том, что JavaScript был сгенерирован компилятором TypeScript. Давайте теперь посмотрим на замыкание, сгенерированное компилятором TypeScript:

```
var DoorState;
(function (DoorState) {
   DoorState[DoorState["Open"] = 0] = "Open";
   DoorState[DoorState["Closed"] = 1] = "Closed";
   DoorState[DoorState["Ajar"] = 2] = "Ajar";
})(DoorState || (DoorState = {}));
```

Этот странно выглядящий синтаксис строит объект, имеющий определенную внутреннюю структуру. Именно эта внутренняя структура позволяет нам использовать данное перечисление различными способами, которые мы видели. Если мы исследуем данную структуру во время отладки своего кода JavaScript, то увидим, что внутренняя структура объекта `DoorState` выглядит так:

```
DoorState
{...}
   [prototype]: {...}
   [0]: "Open"
   [1]: "Closed"
   [2]: "Ajar"
   [prototype]: []
```

```
Ajar: 2
Closed: 1
Open: 0
```

У объекта DoorState есть свойство с именем "0", которое имеет строковое значение "Open". К сожалению, в JavaScript число 0 не является допустимым именем свойства, поэтому мы не можем получить доступ к этому свойству, просто используя DoorState.0. Вместо этого мы должны получить доступ к этому свойству, используя либо DoorState[0], либо DoorState["0"]. Доступ к значению этого свойства вернет строковое значение "Open". Обратите внимание, что у объекта DoorState также есть свойство с именем Open, для которого задано числовое значение 0. Слово Open является допустимым именем свойства в JavaScript, поэтому мы можем получить доступ к этому свойству с помощью DoorState["Open"] или просто DoorState.Open, который соответствует тому же свойству JavaScript.

Обратите внимание, что внутренняя структура строковых перечислений не имеет свойств на основе чисел, поэтому мы не можем использовать синтаксис enum [propertyIndex] при использовании строковых перечислений.

Const

Язык TypeScript также позволяет определять переменную как константу, используя ключевое слово const. Если переменная была помечена как const, то ее значение может быть установлено только после определения переменной и не может быть изменено впоследствии. Рассмотрим приведенный ниже код:

```
const constValue = "test";
constValue = "updated";
```

Здесь мы определили переменную с именем constValue и указали, что ее нельзя изменить с помощью ключевого слова const. Попытка скомпилировать этот код приведет к следующей ошибке компиляции:

```
error TS2450: Left-hand side of assignment expression
cannot be a constant or a read-only property.
```

Эта ошибка фактически генерируется для второй строки кода, где мы пытаемся изменить значение переменной constValue на строку "updated".

Ключевое слово let

Переменные в JavaScript определяются с помощью ключевого слова var. К сожалению, среда выполнения JavaScript очень мягко реагирует на то, где встречаются эти определения, и позволит использовать переменную до того, как она

будет определена. Если среда выполнения JavaScript встречает переменную, которая ранее не была определена или ей было присвоено значение, тогда значение этой переменной будет неопределенным (undefined). Рассмотрим следующий код:

```
console.log(`anyValue = ${anyValue}`);
var anyValue = 2;
console.log(`anyValue = ${anyValue}`);
```

Здесь мы начнем с записи значения переменной с именем anyValue в консоль. Однако обратите внимание, что эта переменная определяется только во второй строке данного фрагмента кода. Другими словами, мы можем использовать переменную в JavaScript до того, как она будет определена. Вывод этого кода выглядит так:

```
anyValue = undefined
anyValue = 2
```

Семантика использования ключевого слова var ставит перед нами небольшую проблему. В ходе использования этого ключевого слова не проверяется, была ли определена сама переменная, прежде чем мы на самом деле ее используем. Очевидно, это может привести к нежелательному поведению, так как значение неопределенной или незанятой переменной всегда равно undefined.

TypeScript вводит ключевое слово let, которое можно использовать вместо ключевого слова var при определении переменных. Одним из преимуществ использования ключевого слова let является то, что мы не можем использовать имя переменной до ее определения. Рассмотрим следующий код:

```
console.log(`letValue = ${lValue}`);
let lValue = 2;
```

Здесь мы пытаемся записать значение переменной lValue в консоль, еще до того, как она была определена, подобно тому, как мы использовали переменную anyValue ранее. Однако при использовании ключевого слова let вместо var этот код выдаст ошибку:

```
error TS2448: Block-scoped variable 'lValue' used before its
declaration.
```

Чтобы исправить этот код, нам нужно определить нашу переменную lValue перед ее первым использованием, как показано ниже:

```
let lValue = 2;
console.log(`lValue = ${lValue}`);
```

Этот код будет правильно скомпилирован и выведет на консоль следующее:

```
lValue = 2
```

Еще одним побочным эффектом использования ключевого слова `let` является то, что переменные, определенные с помощью `let`, имеют область видимости на уровне блока. Это означает, что их значение и определение ограничены блоком кода, в котором они находятся. В качестве примера рассмотрим следующий код:

```
let lValue = 2;
console.log(`lValue = ${lValue}`);
if (lValue == 2) {
   let lValue = 2001;
   console.log(`block scoped lValue : ${lValue}`);
}
console.log(`lValue = ${lValue}`);
```

Здесь мы определяем переменную `lValue` в первой строке, используя ключевое слово `let`, и присваиваем ей значение 2. Затем мы записываем значение `lValue` в консоль. В первой строке, где идет оператор `if`, обратите внимание, что мы переопределяем переменную с именем `lValue` для хранения значения `2001`. Затем мы записываем значение `lValue` в консоль (в блоке оператора `if`). Последняя строка этого фрагмента кода снова записывает значение переменной `lValue` в консоль, но на этот раз `lValue` находится за пределами области блока оператора `if`. Вывод этого кода выглядит так:

```
lValue = 2
block scoped lValue : 2001
lValue = 2
```

Эти результаты показывают, что переменные `let` ограничены областью, в которой они определены. Другими словами, `let lValue = 2001;` оператор определяет новую переменную, которая будет видна только внутри блока оператора `if`. Поскольку это новая переменная, она также не будет влиять на значение переменной `lValue`, которая находится за ее пределами. Вот почему значение `lValue` равно 2 как до, так и после блока оператора `if` и `2001` внутри него.

Таким образом, оператор `let` предоставляет более безопасный способ объявления переменных и ограничения их допустимости в соответствии с текущей областью действия.

Компилятор TypeScript будет автоматически генерировать ошибки для переменных, которые использовались до того, как они были определены, если мы используем опцию `--strictNullChecks` в нашем файле `tsconfig.json`. Мы будем рассматривать эти параметры компилятора более подробно в `главе 5` *«Файлы объявлений и строгие опции компилятора»*.

Определенное присваивание

Однако в некоторых случаях нам бы хотелось объявить переменную с помощью ключевого слова let и использовать ее до того, как TypeScript решит, что она определена. Рассмотрим следующий код:

```
let globalString: string;
setGlobalString();
console.log(`globalString : ${globalString}`);
function setGlobalString() {
   globalString = "this has been set";
}
```

В данном случае мы объявили переменную с именем `globalString`. Затем мы вызываем функцию с именем `setGlobalString`, которая устанавливает значение переменной `globalString`. Таким образом, согласно логике этого кода, переменная `globalString` не является неопределенной и должна содержать значение на этом этапе. Следующая строка кода записывает значение `globalString` в консоль.

К сожалению, компилятор TypeScript не следует и не может следовать нашему пути выполнения, чтобы проверить, использовалась ли конкретная переменная, прежде чем она получит значение. Что касается компилятора, мы объявили переменную `globalString`, но не присвоили ей значение до того, как она будет использоваться в `console.log`. Компиляция этого кода приведет к следующей ошибке:

```
error TS2454: Variable 'globalString' is used before being
assigned.
```

В интересах данного сценария мы можем использовать синтаксис утверждения присваивания, который заключается в добавлении восклицательного знака (`!`) после переменной, которая уже была присвоена. Есть два места, где можно сделать это, чтобы наш код компилировался.

Сначала при объявлении переменной:

```
let globalString!: string;
```

Здесь мы добавили восклицательный знак к объявлению переменной `globalString`. Это говорит компилятору, что мы утверждаем, что переменная была назначена и что она не должна вызывать ошибку, если считает, что она использовалась до назначения.

Второе место, где мы можем использовать утверждение присваивания, – там, где переменная фактически используется, поэтому мы могли бы изменить нашу функцию `console.log` следующим образом:

```
console.log(`globalString : ${globalString!}`);
```

В данном случае мы применили утверждение присваивания, в котором переменная `globalString` используется путем добавления восклицательного знака к переменной.

Хотя у нас есть возможность нарушать стандартные правила TypeScript с помощью утверждений присваивания, более важный вопрос: почему? Почему мы должны структурировать наш код таким образом? Почему мы используем глобальную переменную в первую очередь? Почему бы не использовать другие методы ООП, чтобы избежать этого сценария? Мы рассмотрим ориентацию объекта в следующей главе и посмотрим, как можно использовать наследование и интерфейсы, чтобы найти более надежный способ решения этих проблем.

Типы свойств с точечной нотацией

При работе с объектами JavaScript довольно часто их определяют с помощью имен свойств, которые являются строками, как видно из приведенного ниже кода:

```
let normalObject = {
   id : 1,
   name : "test" }
let stringObject = {
   "testProperty": 1,
   "anotherProperty": "this is a string"
}
```

В данном случае мы определили объект с именем `normalObject` и свойствами `id` и `name`. Мы можем получить доступ к этим свойствам, обращаясь к `normalObject.id` или `normalObject.name`, как мы уже видели неоднократно. Обратите внимание, однако, на незначительное изменение в определении объекта `stringObject`. Здесь каждое имя свойства заключается в двойные кавычки (""), что делает его строковым свойством. Мы можем обращаться к этим строковым свойствам двумя способами:

```
let testProperty = stringObject.testProperty;
console.log(`testPropertyValue = ${testProperty}`);
let testStringProperty = stringObject["testProperty"];
console.log(`"testPropertyValue" = ${testStringProperty}`);
```

Здесь мы используем синтаксис `stringObject.testProperty` и синтаксис `stringObject["testProperty"]` для доступа к свойству `testProperty`. Для TypeScript это одно и то же, как показано в следующем фрагменте кода:

```
testPropertyValue = 1
"testPropertyValue" = 1
```

Числовые разделители

TypeScript также включает в себя поддержку стандарта ECMAScript, позволяющего использовать подчеркивание (_) при определении больших чисел. Рассмотрим следующий код:

```
let oneMillion = 1_000_000;
console.log(`oneMillion = ${oneMillion}`);
```

Здесь мы определили числовую переменную с именем `oneMillion` со значением 1 миллион. Обратите внимание на использование символа _ для более удобочитаемого представления такого большого числа. Когда TypeScript выдаст эквивалентный код JavaScript, он преобразует это удобочитаемое число в десятичный эквивалент, поэтому вывод этого кода будет таким:

```
oneMillion = 1000000;
```

Данные числовые разделители могут также использоваться для других чисел, таких как шестнадцатеричные значения:

```
let limeGreenColor = 0x00_FF_00;
console.log(`limeGreenColor = ${limeGreenColor}`);
```

Здесь мы определили числовое значение `limeGreenColor` и присвоили ему шестнадцатеричное значение `00FF00`. Еще раз обратите внимание, что разделители делают это число более читабельным и что полученный код JavaScript будет содержать десятичный эквивалент, что означает, что вывод этого кода будет таким:

```
limeGreenColor = 65280
```

Функции

До сих пор мы изучали, как добавлять аннотации типов к переменным и объектам. Тем не менее этот простой синтаксис также может использоваться с функциями для обеспечения безопасности типов всякий раз, когда используется функция. Давайте теперь рассмотрим синтаксис аннотации типов применительно к функциям и изучим эти правила более подробно.

Типы возвращаемого значения

С помощью очень простого синтаксического сахара, который использует TypeScript, мы можем легко определить тип переменной, которую должна возвращать функция. Другими словами, когда мы вызываем функцию и она возвращает значение, к какому типу следует относить тип возвращаемого значения?

Рассмотрим следующий код:

```
function addNumbers(a: number, b: number) : string {
   return a + b;
}

var addResult = addNumbers(2,3);
console.log(`addNumbers returned : ${addResult}`);
```

Здесь мы добавили тип : `number` к обоим параметрам функции `addNumbers` (`a` и `b`), а также добавили тип : `string` сразу после фигурных скобок `()` в определении функции. Размещение аннотации типа после определения функции означает, что мы определяем тип возвращаемого значения всей функции. В нашем примере тип возвращаемого значения функции `addNumbers` должен быть `string`.

К сожалению, этот код выдает сообщение об ошибке:

```
error TS2322: Type 'number' is not assignable to type 'string'
```

Это сообщение указывает на то, что тип возвращаемого значения функции `addNumbers` должен быть строкой (`string`). При более внимательном рассмотрении кода мы заметим, что код, вызывающий нарушение, на самом деле возвращает `a + b`. Поскольку `a` и `b` являются числами, мы возвращаем результат сложения двух чисел, тип которого – `number`. Чтобы исправить этот код, нам нужно убедиться, что функция возвращает строку:

```
function addNumbers(a: number, b: number) : string {
   return (a + b).toString();
}
```

Этот код теперь будет скомпилирован правильно и выведет следующее:

```
addNumbers returned : 5
```

Анонимные функции

В языке JavaScript также существует концепция анонимных функций. Это функции, которые определяются на лету и не указывают имя функции. Рассмотрим приведенный ниже код JavaScript:

```
var addVar = function(a,b) {
   return a + b;
}
var addVarResult = addVar(2,3);
console.log("addVarResult:"" + addVarResult);
```

Здесь мы определяем функцию, у которой нет имени и которая добавляет два значения. Поскольку у функции нет имени, она известна как анонимная функ-

ция. Эта анонимная функция затем присваивается переменной с именем `addVar`. Переменная `addVar` может быть вызвана как функция с двумя параметрами, и возвращаемое значение будет результатом выполнения анонимной функции. Вывод этого кода будет таким:

```
addVarResult:5
```

Давайте теперь перепишем предыдущую анонимную функцию JavaScript в TypeScript:

```
var addFunction = function(a:number, b:number) : number {
   return a + b;
}

var addFunctionResult = addFunction(2,3);
console.log(`addFunctionResult : ${addFunctionResult}`);
```

Здесь видно, что TypeScript разрешает анонимные функции так же, как JavaScript, но еще допускает стандартные аннотации типов. Вывод этого кода такой:

```
addFunctionResult : 5
```

Необязательные параметры

Когда мы вызываем функцию JavaScript, которая ожидает параметры, и не предоставляем их, значение параметра в функции будет неопределенным. В качестве примера рассмотрим следующий код JavaScript:

```
var concatStrings = function(a,b,c) {
   return a + b + c;
}

var concatAbc = concatStrings("a", "b", "c");
console.log("concatAbc :" + concatAbc);

var concatAb = concatStrings("a", "b");
console.log("concatAb :" + concatAb);
```

Вывод этого кода выглядит так:

```
concatAbc :abc
concatAb :abundefined
```

Здесь мы определили функцию `concatStrings`, которая принимает три параметра `a`, `b` и `c` и просто возвращает сумму этих значений. Затем мы вызываем эту функцию с тремя аргументами и присваиваем результат переменной `concatAbc`. Как видно, на выходе она возвращает строку `"abc"`, как и ожидалось. Однако если

мы предоставляем только два аргумента, как видно при использовании переменной `concatAb`, функция возвращает строку "`abundefined`". В JavaScript, если мы вызываем функцию и не предоставляем параметр, отсутствующий параметр будет неопределенным. В данном случае это параметр `c`.

TypeScript использует синтаксис знака вопроса ? для указания необязательных параметров. Это позволяет имитировать синтаксис вызова JavaScript, где мы можем вызывать одну и ту же функцию с некоторыми отсутствующими аргументами. В качестве примера рассмотрим следующий код:

```
function concatStrings( a: string, b: string, c?: string) {
   return a + b + c;
}

var concat3strings = concatStrings("a", "b", "c");
console.log(`concat3strings : ${concat3strings}`);

var concat2strings = concatStrings("a", "b");
console.log(`concat2strings : ${concat2strings}`);

var concat1string = concatStrings("a");
```

Здесь у нас есть сильно типизированная версия оригинальной JavaScript-функции `concatStrings`, которую мы использовали ранее. Обратите внимание на добавление символа ? в синтаксисе для третьего параметра: `c ?: string`. Это указывает на то, что третий параметр является необязательным, и, следовательно, весь предыдущий код будет компилироваться без ошибок, за исключением последней строки. Последняя строка выдаст ошибку:

```
error TS2554: Expected 2-3 arguments, but got 1
```

Эта ошибка возникает, потому что мы пытаемся вызвать функцию `concatStrings` только с одним параметром. Наше определение функции тем не менее требует как минимум двух параметров, при этом только третий параметр является необязательным.

Любые необязательные параметры должны быть последними параметрами, определенными в определении функции. У вас может быть столько необязательных параметров, сколько вы хотите, если обязательные параметры предшествуют необязательным.

Параметры по умолчанию

Тонкий вариант синтаксиса необязательного параметра позволяет указать значение параметра по умолчанию, если оно не указано. Давайте изменим предыдущее определение функции, чтобы использовать необязательный параметр со значением по умолчанию, как показано ниже:

```
function concatStringsDefault(
   a: string,
   b: string,
   c: string = "c") {
      return a + b + c;
}

var defaultConcat = concatStringsDefault("a", "b");
console.log(`defaultConcat : ${defaultConcat}`);
```

Это определение функции теперь отбросило синтаксис необязательного параметра ?, но вместо этого присвоило значение "c" последнему параметру, `c:string = "c"`. Используя параметры по умолчанию, если мы не предоставим значение для параметра с именем c, вместо этого функция `concatStringsDefault` заменит значение по умолчанию "c". Следовательно, аргумент c не будет неопределенным. Поэтому вывод этого кода будет таким:

```
defaultConcat : abc
```

Обратите внимание, что использование значения параметра по умолчанию автоматически делает параметр, который имеет значение по умолчанию, необязательным.

Оставшиеся параметры

Язык JavaScript также позволяет вызывать функцию с переменным числом аргументов. Чтобы проиллюстрировать это, мы можем использовать причудливое свойство языка JavaScript. Каждая функция JavaScript имеет доступ к специальной переменной с именем arguments, которую можно использовать для получения всех аргументов, переданных в функцию. В качестве примера рассмотрим следующий код:

```
function testArguments() {
   if (arguments.length > 0) {
      for (var i = 0; i < arguments.length; i++ ) {
         console.log("argument[" + i + "] = " + arguments[i]);
      }
   }
}
testArguments(1,2,3);
testArguments("firstArg");
```

Здесь мы определили функцию с именем `testArguments`, у которой нет именованных параметров. Обратите внимание, что мы можем использовать специальную переменную с именем arguments, чтобы проверить, была ли функция вызвана с какими-либо аргументами. В нашем примере мы просто перебираем массив

arguments и записываем значение каждого аргумента в консоль, используя индекс массива arguments `[i]`. Вывод этого кода выглядит так:

```
argument[0] = 1
argument[1] = 2
argument[2] = 3
argument[0] = firstArg
```

Чтобы выразить определение эквивалентной функции в TypeScript, нам нужно будет использовать синтаксис, известный как синтаксис оставшихся параметров. Оставшиеся параметры используют синтаксис TypeScript из трех точек (`...`) в объявлении функции, чтобы выразить переменное число параметров функции. Ниже приводится эквивалентная функция `testArguments`, выраженная в TypeScript:

```
function testArguments(... argArray: number []) {
   if (argArray.length > 0) {
      for (var i = 0; i < argArray.length; i++) {
         console.log(`argArray[${i}] = ${argArray[i]}`);
         // use JavaScript arguments variable
         console.log(`arguments[${i}] = ${arguments[i]}`)
      }
   }
}

testArguments(9);
testArguments(1,2,3);
```

Обратите внимание на использование синтаксиса `... argArray: number[]` для нашего параметра функции `testArguments`. Этот синтаксис сообщает компилятору TypeScript, что функция может принимать любое количество аргументов, если каждый аргумент является числом. Поэтому мы можем вызвать эту функцию, как видно из двух последних строк предыдущего кода, с любым количеством числовых значений. В этом цикле for также есть два оператора `console.log`. Первый использует `argArray[i]`, который обращается к имени оставшегося параметра, а второй использует стандартную переменную JavaScript, `arguments [i]`, которая все еще доступна для использования в TypeScript.

Вывод этого кода выглядит так:

```
argArray[0] = 9
arguments[0] = 9
argArray[0] = 1
arguments[0] = 1
argArray[1] = 2
arguments[1] = 2
argArray[2] = 3
arguments[2] = 3
```

Тонкое различие между использованием `argArray` и `arguments` – это выводимый тип аргумента. Поскольку мы явно указали, что `argArray` имеет тип `number`, TypeScript будет рассматривать любой элемент массива `argArray` как число. Однако у внутреннего массива `arguments` нет выводимого типа, и поэтому он будет рассматриваться как тип any.

Мы также можем комбинировать обычные параметры с оставшимися в определении функции, если оставшиеся параметры являются последними, которые должны быть определены в списке:

```
function testNormalAndRestArguments(
   arg1: string,
   arg2, number,
   ...argArray: number[]
) {
}
```

Здесь у нас есть два нормальных параметра с именами `arg1` и `arg2`, а затем оставшийся параметр `argArray`. Ошибочное размещение оставшегося параметра в начале списка параметров приведет к ошибке компиляции.

Функции обратного вызова

Одним из самых мощных свойств JavaScript и фактически технологии, на которой был построен Node, является концепция функций обратного вызова. Функция обратного вызова – это функция, которая передается в другую функцию, а затем обычно вызывается внутри нее. Таким образом достигается асинхронное программирование. Предоставляя функцию обратного вызова, мы говорим функции, которую вызываем, иди и делай то, что тебе нужно, а когда закончишь, вызови эту функцию. Так же, как мы можем передать значение в функцию, мы можем передать функцию в функцию.

Это лучше всего проиллюстрировать, посмотрев на пример кода JavaScript:

```
var callbackFunction = function(text) {
   console.log('inside callbackFunction ' + text);
}

function doSomethingWithACallback( initialText, callback ) {
   console.log('inside doSomethingWithCallback ' + initialText);
   callback(initialText);
}

doSomethingWithACallback('myText'', callbackFunction);
```

Здесь мы начинаем с переменной с именем `callbackFunction`, которая является функцией, принимающей один параметр. Эта функция обратно-

го вызова просто записывает аргумент `text` в консоль. Затем мы определяем функцию с именем `doSomethingWithACallback`, которая принимает два параметра: `initialText` и `callback`. Первая строка этой функции просто записывает `"inside doSomethingWithACallback"` в консоль. Вторая строка `doSomethingWithACallback` – интересный момент. Предполагается, что аргумент `callback` на самом деле является функцией и вызывает ее, передавая переменную `initialText`. Если мы запустим этот код, то получим два сообщения, записанных в консоль:

```
inside doSomethingWithCallback myText
inside callbackFunction myText
```

Эти выходные данные ясно показывают, что мы входим в функцию `doSomethingWithACallback`, записываем сообщение в консоль и затем вызываем функцию `callbackFunction`.

Но что произойдет, если мы допустим ошибку и не передадим функцию в качестве обратного вызова, когда мы должны это сделать? В предыдущем коде нет ничего, что указывало бы на то, что второй параметр `doSomethingWithACallback` должен быть функцией. Если бы мы случайно вызвали функцию `doSomethingWithACallback` с двумя строками, как показано в следующем фрагменте кода:

```
doSomethingWithACallback('myText', 'anotherText');
```

Мы получили бы ошибку времени выполнения JavaScript:

```
TypeError: callback is not a function
```

Однако программисты на JavaScript, придающие большое значение защите, сначала проверяют, действительно ли параметр `callback` является функцией, прежде чем вызывать его:

```
function doSomethingWithACallback( initialText, callback ) {
   console.log('inside doSomethingWithCallback ' + initialText);
   if (typeof callback === "function") {
      callback(initialText);
   } else {
      console.log(initialText + ' is not a function !!')
   }
}
doSomethingWithACallback('myText'', ''anotherText'');
```

Обратите внимание на третью строку этого фрагмента кода, где мы проверяем природу переменной `callback`, перед тем как вызвать ее. Если это не функция, мы записываем сообщение в консоль. Вывод фрагмента кода будет таким:

```
inside doSomethingWithCallback myText
anotherText is not a function !!
```

Поэтому программисты на JavaScript должны быть осторожны при работе с обратными вызовами. Во-первых, они должны кодировать недопустимое использование функций обратного вызова, а во-вторых, должны документировать и понимать, какие параметры фактически являются обратными вызовами.

Что, если бы мы могли задокументировать наши функции обратного вызова JavaScript в своем коде, а затем предупредить пользователей, если они не передают функцию, когда она ожидается? Аннотации типов для функций обратного вызова сделают это за нас и будут выдавать ошибки компиляции, если мы нарушим эти правила. Давайте посмотрим, как это делается в TypeScript.

Сигнатуры функций

Синтаксический сахар TypeScript, который обеспечивает сильную типизацию для обычных переменных, также может использоваться с функциями обратного вызова. Для этого TypeScript использует новый синтаксис под названием стрелочный синтаксис, () =>. Его использование означает, что один из параметров функции должен быть другой функцией. Давайте подробнее рассмотрим, что это значит. Мы перепишем предыдущий пример обратного вызова JavaScript в TypeScript следующим образом:

```
function callbackFunction(text: string) {
  console.log(`inside callbackFunction ${text}`);
}
```

Мы начнем с начальной функции обратного вызова, которая принимает один текстовый параметр и записывает сообщение в консоль при вызове этой функции. Затем мы можем определить функцию `doSomethingWithACallback` следующим образом:

```
function doSomethingWithACallback(
  initialText: string,
  callback : (initialText: string) => void
) {
  console.log(`inside doSomethingWithCallback ${initialText}`);
  callback(initialText);
}
```

Здесь мы определили нашу функцию `doSomethingWithACallback` с двумя параметрами. Первый параметр – `initialText` типа `string`. Второй параметр называется `callback` и теперь использует стрелочный синтаксис, чтобы указать, что этот параметр должен быть функцией. Давайте посмотрим на этот синтаксис подробнее:

```
callback: (initialText: string) => void
```

Используемый здесь аргумент `callback` типизирован (с помощью синтаксиса :) как функция при помощи стрелочного синтаксиса `() =>`. Кроме того, эта функция принимает параметр с именем `initialText` типа `string`. Справа от стрелочного синтаксиса мы видим новый базовый тип TypeScript, `void`. `void` – это ключевое слово, которое используется, чтобы обозначить, что функция не возвращает значение.

Таким образом, функция `doSomethingWithACallback` будет принимать в качестве второго аргумента только функцию, которая принимает один строковый параметр и возвращает `void`.

Затем мы можем использовать эту функцию следующим образом:

```
doSomethingWithACallback("myText"", callbackFunction);
```

Этот фрагмент кода такой же, как был использован в нашем примере с JavaScript ранее. TypeScript проверит тип параметра `callbackFunction`, который был передан, и убедится, что это на самом деле функция, которая принимает одну строку в качестве аргумента и ничего не возвращает. Если мы попытаемся вызвать `doSomethingWithACallback` неправильно, скажем, с двумя строками:

```
doSomethingWithACallback("myText", "this is not a function");
```

компилятор выдаст следующее сообщение:

```
error TS2345: Argument of type '"this is not a function"' i
s not assignable to parameter of type '(initialText: string)
=> void'.
```

В этом сообщении четко указано, что второй аргумент, `"this is not a function"`, не является функцией типа `(initialText: string) => void`, как и ожидалось.

Учитывая эту сигнатуру функции для параметра `callback`, приведенный ниже код также будет генерировать ошибки времени компиляции:

```
function callbackFunctionWithNumber(arg1: number) {
   console.log(`inside callbackFunctionWithNumber ${arg1}`)
}
doSomethingWithACallback("myText"", callbackFunctionWithNumber);
```

Здесь мы определяем функцию с именем `callBackFunctionWithNumber`, которая принимает число в качестве единственного параметра. При попытке скомпилировать этот код мы получим сообщение об ошибке, указывающее на то, что параметр `callback`, который теперь является функцией `callBackFunctionWithNumber`, также не имеет правильной сигнатуры:

```
error TS2345: Argument of type '(arg1: number) => void' is not
assignable to parameter of type '(initialText: string) => void'.
```

В сообщении четко указано, что ожидается параметр типа `(initialText: string)=> void`, но вместо этого использовался аргумент типа `(arg1: number) => void`.

В сигнатурах функций имя параметра (`arg1` или `initialText`) не должно быть одинаковым. Только количество параметров, их типы и тип возвращаемого значения функции должны быть одинаковыми.

Это очень мощное свойство TypeScript – определение в коде того, какими должны быть сигнатуры функций, и предупреждение пользователей о том, когда они не вызывают функцию с правильными параметрами. Определение типов функций также позволяет интегрированной среде разработки, в которой мы работаем, предлагать возможности Intellisense, которые будут уведомлять нас о правильном использовании конкретной функции.

Как мы знаем из введения в TypeScript, это также может быть важно, когда мы работаем со сторонними библиотеками. Вместо того чтобы хранить копию документации под рукой при работе с библиотекой, мы можем воспользоваться возможностями Intellisense более эффективно. Однако прежде чем мы сможем использовать сторонние функции, классы или объекты в TypeScript, нам нужно определить их сигнатуры функций. Эти определения функций помещены в особый тип файла TypeScript, файл объявления, и сохраняются с расширением `.d.ts`. Мы подробно рассмотрим файлы объявлений в `главе 5` *«Файлы объявлений и строгие опции компилятора»*.

Переопределение функций

Поскольку JavaScript – это динамический язык, мы часто можем вызывать одну и ту же функцию с разными типами аргументов. Рассмотрим следующий код:

```
function add(x,y) {
   return x + y; }
console.log('add(1,1)=' + add(1,1));
console.log('add("1","1")=' + add("1","1"));
```

Здесь мы определяем простую функцию `add`, которая возвращает сумму двух ее параметров, `x` и `y`. Затем мы просто записываем результат этой функции с разными типами – двумя числами и двумя строками. Если мы выполним приведенный выше код, мы увидим следующий вывод:

```
add(1,1)=2
add("1","1")=11
```

Чтобы воспроизвести возможность вызова одной и той же функции с разными типами, TypeScript использует специальный синтаксис под названием переопределение функций. Если бы мы повторили предыдущий код в TypeScript, нам нужно было бы использовать следующий синтаксис переопределения функции:

```
function add(a: string, b: string) : string;
function add(a: number, b: number) : number;
function add(a: any, b: any): any {
   return a + b;
}
console.log(`add(1,1)= ${add(1,1)}`);
console.log(`add("1","1")= ${add("1","1")}`);
```

Здесь мы указываем сигнатуру переопределения функции для функции `add`, которая принимает две строки и возвращает строку. Затем мы указываем второе переопределение функции, которое использует то же имя функции, но использует числа в качестве параметров. За этими двумя переопределениями далее следует фактическое тело функции. Последние две строки этого фрагмента вызывают функцию `add`, сначала с двумя числами, а затем с двумя строками. Вывод этого кода выглядит так:

```
add(1,1)= 2
add("1","1")= 11
```

Тут есть три интересных момента. Во-первых, ни одна из сигнатур функций в первых двух строках фрагмента кода фактически не имеет тела функции. Во-вторых, окончательное определение функции использует спецификатор типа any и в конечном итоге включает тело функции. Чтобы переопределить функции таким образом, мы должны следовать этому соглашению, и конечная сигнатура (которая включает в себя тело функции) должна использовать тип спецификатора any, так как все остальное будет генерировать ошибки во время компиляции.

Последнее, на что следует обратить внимание, – это то, что хотя в конечном теле функции используется тип any, данная сигнатура, по существу, скрыта с помощью этого соглашения. На самом деле мы ограничиваем функцию `add` только двумя строками или двумя числами. Если мы вызываем функцию с двумя логическими значениями:

```
console.log(`add(true,false)= ${add(true,false)}`);
```

TypeScript будет выдавать ошибки компиляции:

```
error TS2345: Argument of type 'true' is not assignable to
parameter of type 'number'.
```

Try...catch

TypeScript позволяет использовать блоки `try...catch` так же, как это делает JavaScript. Рассмотрим следующий код:

```
try {
   console.log(`1. attempting to parse JSON`);
```

```
    JSON.parse(`abcd=234`);
} catch (error) {
    console.log(`2. try catch error : ${error}`);
} finally {
    console.log(`3. finally`);
}
```

Здесь, в блоке try, мы записываем сообщение в консоль, а затем пытаемся вызвать функцию, которая может выдавать или не выдавать ошибку. Функция `JSON.parse` выдаст ошибку, если она не сможет проанализировать входную строку как допустимый JSON. В этом блоке кода мы умышленно вызываем ошибку, используя недопустимую строку JSON. Если выброшена ошибка, будет выполнен блок `catch`, а затем – блок `finally`, независимо от того, была ли выброшена ошибка. Вывод этого фрагмента кода будет таким:

```
1. attempting to parse JSON
2. try catch error : SyntaxError: Unexpected token a in JSON
   at position 0
3. finally
```

Этот вывод, как и ожидалось, записывает сообщения в консоль по порядку.

Мы также можем опустить параметр error из блока `catch`, если нам не интересно, какая ошибка была выброшена:

```
try {
    console.log(`1. attempting to parse JSON`);
    JSON.parse(`abcd=234`);
} catch {
    console.log(`2. caught`);
} finally {
    console.log(`3. finally`);
}
```

Вывод этого кода будет таким:

```
1. attempting to parse JSON
2. caught
3. finally
```

Расширенные типы

TypeScript также обладает рядом расширенных возможностей языка, которые можно использовать при работе с базовыми типами и объектами. Они позволяют немного больше смешивать и сопоставлять типы, а также создавать новые, которые являются комбинациями других типов. В этом разделе мы кратко рассмотрим возможности расширенных типов, включая:

- объединенные типы;
- охранников типов;
- псевдонимы типов;
- Null и undefined;
- Never и unknown;
- Object rest and spread;
- кортежи;
- Bigint.

Объединенные типы

TypeScript позволяет выражать тип как комбинацию двух или более типов. Эти типы известны как объединенные типы, и их объявление использует символ вертикальной черты (`|`), чтобы перечислить все типы, которые будут составлять новый тип. Рассмотрим следующий код:

```
var unionType : string | number;
unionType = 1;
console.log(`unionType : ${unionType}`);
unionType = "test";
console.log(`unionType : ${unionType}`);
```

Здесь мы определили переменную с именем `unionType`, которая использует синтаксис объединенного типа для обозначения того, что она может содержать строку или число. Затем мы присваиваем этой переменной число и записываем ее значение в консоль. После этого мы присваиваем строку этой переменной и снова записываем ее значение в консоль. Данный фрагмент кода выведет следующее:

```
unionType : 1
unionType : test
```

Поначалу использование объединенных типов может показаться немного странным. В конце концов, зачем простой переменной содержать и число, и строку? Конечно, это в какой-то степени нарушает наши правила сильной типизации. Однако преимущества объединенных типов можно увидеть при использовании функций или при вызове конечных точек службы REST, которые могут возвращать разные структуры ответа.

Охранники типов

При работе с объединенными типами компилятор по-прежнему будет применять правила сильной типизации для обеспечения безопасности типов. В качестве примера рассмотрим следующий код:

```
function addWithUnion(
```

```
    arg1 : string | number,
    arg2 : string | number
) {
    return arg1 + arg2;
}
```

Здесь мы определяем функцию с именем `addWithUnion`, которая принимает два параметра и возвращает их сумму. Аргументы `arg1` и `arg2` являются объединенными типами и поэтому могут быть либо строкой, либо числом. Однако при компиляции этого кода появится ошибка:

```
error TS2365: Operator '+' cannot be applied to types
'string | number' and 'string | number'.
```

В данном случае компилятор говорит нам, что в теле функции, где он пытается добавить `arg1` к `arg2`, он не может сказать, какой тип `arg1` вот-вот попытается использовать его. Это строка или число?

Вот тут и вступают в действие охранники типов. Охранник типов – это выражение, которое выполняет проверку нашего типа, а затем гарантирует этот тип в своей области видимости. Рассмотрим следующий код:

```
function addWithTypeGuard(
    arg1 : string | number,
    arg2 : string | number
    ) : string | number {
        if( typeof arg1 === "string") {
        // в рамках этого кода arg1 рассматривается как строка
        console.log('первый аргумент - это строка');
        return arg1 + arg2;
    }
    if (typeof arg1 === "number" && typeof arg2 === "number") {
        // в рамках этого кода arg1 и arg2 рассматриваются как числа
        console.log('оба аргумента - числа');
        return arg1 + arg2;
    }
    console.log('default return');
    return arg1.toString() + arg2.toString();
}
```

Здесь у нас есть функция с именем `addWithTypeGuard`, которая принимает два аргумента и использует наш синтаксис объединенного типа с целью указать, что `arg1` и `arg2` могут быть либо строкой, либо числом.

В теле кода у нас есть два оператора `if`. Первый оператор проверяет, является ли тип аргумента `arg1` строкой. Если это строка, то тип `arg1` рассматривается как строка в теле оператора `if`. Второй оператор проверяет, имеют ли `arg1` и `arg2`

тип number. В теле второго оператора `if` и `arg1`, `arg2` рассматриваются как числа. Эти два оператора являются охранниками типов.

Обратите внимание, что наш последний оператор return вызывает функцию toString для `arg1` и `arg2`. У всех основных типов JavaScript по умолчанию есть функция `toString`, поэтому мы, по сути, рассматриваем оба аргумента как строки и возвращаем результат. Давайте посмотрим, что происходит, когда мы вызываем эту функцию с различными комбинациями типов:

```
console.log(`addWithTypeGuard(1,2)= ${addWithTypeGuard(1,2)}`);
```

Здесь мы вызываем функцию с двумя числами и получаем следующее:

```
both arguments are numbers addWithTypeGuard(1,2) = 3
```

Это показывает, что код удовлетворил второго оператора `if`. Если мы вызываем функцию с двумя строками:

```
console.log(`addWithTypeGuard("1", "2") =
${addWithTypeGuard("1", "2")}`) ;
```

здесь видно, что первый оператор `if` удовлетворен:

```
first argument is a string addWithTypeGuard("1","2") = 12
```

Наконец, когда мы вызываем функцию с номером и строкой:

```
console.log(`addWithTypeGuard(1, "2") =
${addWithTypeGuard(1, "2"")}`) ;
```

В этом случае оба наших оператора охранников типов возвращают значение `false`, поэтому наш возвращаемый код по умолчанию выглядит так:

```
default return
addWithTypeGuard(1,"2")= 12
```

Таким образом охранники типов позволяют проверять тип переменной в вашем коде и затем гарантируют, что тип переменной соответствует вашим ожиданиям.

Псевдонимы типов

Иногда при использовании объединенных типов бывает трудно вспомнить, какие типы разрешены. В ответ на это TypeScript вводит понятие псевдонима типа, где мы можем создать специальный именованный тип для объединения типов. Поэтому псевдоним типа – это удобное соглашение по именованию для объединенных типов. Псевдонимы типов могут использоваться везде, где используются обычные типы. Они обозначаются с помощью ключевого слова `type`. Поэтому мы можем упростить использование объединенных типов в своем коде следующим образом:

```
type StringOrNumber = string | number;
function addWithAlias(
   arg1 : StringOrNumber,
   arg2 : StringOrNumber
) {
   return arg1.toString() + arg2.toString();
}
```

Здесь мы определили псевдоним типа с именем `StringOrNumber`, который является объединенным типом и может быть либо строкой, либо числом. Затем мы используем псевдоним этого типа в своей сигнатуре функции, чтобы и `arg1`, и `arg2` были либо строкой, либо числом.

Интересно, что псевдонимы типов также могут использоваться для сигнатур функций:

```
type CallbackWithString = (string) => void;
function usingCallbackWithString(callback: CallbackWithString) {
   callback("this is a string"); }
```

В данном случае мы определили псевдоним типа с именем `CallbackWithString`. Он представляет собой функцию, которая принимает один строковый параметр и возвращает `void`. Наша функция `usingCallbackWithString` принимает этот псевдоним типа (который является сигнатурой функции) в качестве типа аргумента `callback`.

Когда нам необходимо часто использовать объединенные типы в своем коде, псевдонимы типов предоставляют более простой и интуитивно понятный способ объявления именованных типов объединения.

Null и undefined

В JavaScript, если переменная была объявлена, но ей не присвоено значение, то запрос ее значения вернет undefined. JavaScript также включает ключевое слово `null`, чтобы различать случаи, когда переменная известна, но не имеет значения (`null`), и когда она не была определена в текущей области (`undefined`). Рассмотрим приведенный ниже код:

```
function testUndef(test) {
   console.log('test parameter :' + test);
}
testUndef();
testUndef(null);
```

Здесь мы определили функцию с именем `testUndef`, которая принимает один аргумент с именем `test`. В рамках этой функции мы просто записываем значение в консоль. Затем мы вызываем ее двумя разными способами.

В первом вызове функции `testUndef` нет никаких аргументов. По сути, мы вызываем функцию, не зная или не заботясь о том, какие аргументы ей нужны.

Помните, что аргументы функций JavaScript являются необязательными. В этом случае значение аргумента test в функции `testUndef` будет неопределенным (`undefined`), и результат будет следующим:

```
test parameter :undefined
```

Второй вызов функции `testUndef` передает значение `null` в качестве первого аргумента. Это в основном говорит о том, что мы знаем, что функция нуждается в аргументе, но мы решили вызвать ее без значения. Результат этого вызова будет таким:

```
test parameter :null
```

TypeScript включил два ключевых слова, которые мы можем использовать в этом случае, `null` и `undefined`. Давайте перепишем эту функцию в TypeScript:

```
function testUndef(test : null | number) {
   console.log('test parameter :' + test);
}
```

Здесь мы определили функцию `testUndef`, тем самым позволяя вызвать ее со значениями `null` или `number`. Если мы попытаемся вызвать эту функцию в TypeScript без каких-либо аргументов, как мы это делали в JavaScript:

```
testUndef();
```

TypeScript выдаст ошибку:

```
error TS2554: Expected 1 arguments, but got 0.
```

Ясно, что компилятор TypeScript гарантирует, что мы вызываем функцию `testUndef` с числом либо нулем. Он не позволит нам вызвать ее без аргументов.

Такая возможность указать, что функция может быть вызвана со значением `null`, позволяет гарантировать, что правильное использование нашей функции известно во время компиляции.

Точно так же мы можем определить объект, чтобы разрешить неопределенные (`undefined`) значения:

```
let x : number | undefined;
x = 1;
x = undefined;
x = null;
```

Здесь мы определили переменную с именем `x`, которая может содержать либо число, либо неопределенное значение. После чего мы пытаемся присвоить значения `1`, `undefined` и `null` этой переменной. Компиляция этого кода приведет к ошибке:

```
error TS2322: Type 'null' is not assignable to type 'number
| undefined'.
```

Поэтому компилятор TypeScript защищает наш код, чтобы гарантировать, что переменная x может содержать только число или неопределенное (`undefined`) значение, не позволяя ей содержать нулевое (`null`) значение.

Нулевые операнды

TypeScript также проверяет наличие нулевых или неопределенных значений, когда мы используем базовые операнды, такие как сложение, умножение, меньше чем, деление по модулю и степень. Лучше всего это можно увидеть на простом примере:

```
function testNullOperands(arg1: number, arg2: number | null
| undefined) {
   let a = arg1 + arg2;
   let b = arg1 * arg2;
   let c = arg1 < arg2;
}
```

Здесь мы определили функцию с именем `testNullOperands`, которая принимает два аргумента, `arg1` и `arg2`. Они могут быть числами, нулями или иметь неопределенное значение. Попытка скомпилировать этот код вызовет ряд ошибок, а именно:

```
(92,20): error TS2533: Object is possibly 'null' or 'undefined'
(93,20): error TS2533: Object is possibly 'null' or 'undefined'
(94,20): error TS2533: Object is possibly 'null' or 'undefined'
```

TypeScript выдал ошибку для всех трех наших попыток вычислений. Это очень мощный дополнительный источник проверки типов, и ключом к этим ошибкам является то, что аргумент `arg2` может быть допустимым числом, нулевым или неопределенным значением. Оператор сложения (+) не позволит нам попытаться добавить ноль к числу или неопределенное число, отсюда первая ошибка. Оператор умножения (*) и оператор меньше чем (<) имеют те же ограничения, которые вызывают вторую и третью ошибки.

TypeScript определит, где мы используем операнды, и убедится, что обе стороны операндов являются действительными числами.

Never

TypeScript вводит тип для указания случаев, когда что-либо никогда не должно происходить. Типичный пример – когда функция всегда будет выдавать ошибку и никогда не будет возвращать значение как таковая. Рассмотрим следующий код:

```
function alwaysThrows() {
   throw "this will always throw";
   return -1; }
```

Здесь мы определили простую функцию с именем `alwaysThrows`. Первая строка этой функции выдает ошибку, и как таковая вторая строка данной функции никогда не будет выполнена. Это допустимый код TypeScript, но он указывает на наличие недостатка в нашей логике, поскольку оператор `return` никогда не будет выполнен. Мы можем обезопасить себя, используя возвращаемый тип `never` в определении функции:

```
function alwaysThrows(): never {
   throw "this will always throw";
   return -1; }
```

Здесь мы использовали ключевое слово `never`, чтобы указать, что эта функция никогда не вернет значение. Компилятор теперь выдаст ошибку:

```
error TS2322: Type '-1' is not assignable to type 'never'
```

В сообщении четко указано, что функция, которая никогда не должна возвращаться, пытается вернуть значение –1. Мы четко указали конечный эффект нашей функции, который заключается в том, что она никогда ничего не вернет, и компилятор помогает нам обеспечить это.

Гораздо более практичным использованием ключевого слова never является устранение недостатков в логике нашего кода, которые, как мы знаем, никогда не должны возникать. В качестве примера рассмотрим приведенный ниже код:

```
enum TestNeverEnum {
   FIRST,
   SECOND
}

function getEnumValue(value: TestNeverEnum): string {
   switch (value) {
      case TestNeverEnum.FIRST: return "First case";
   }
   let returnValue: never = value;
}
```

Мы определили перечисление с именем TestNeverEnum, у которого два значения, FIRST и SECOND. Затем мы определяем функцию с именем getEnumValue, которая предназначена для возврата строки на основе переданного значения перечисления. После оператора switch мы определяем значение с именем returnValue, типа never, и присваиваем ему аргумент входящего значения. Этот код выдаст ошибку компиляции:

```
error TS2322: Type 'TestNeverEnum.SECOND' is not assignable
to type 'never'
```

Здесь происходит следующее: наш оператор switch не обрабатывает ветвь, содержащую значение TestNeverEnum.SECOND. Этот недостаток логики означает, что поток кода будет фактически уменьшен до строки let returnValue: never = value. Поскольку здесь мы не использовали never в качестве типа, компилятор выдает ошибку. Этот код необходимо исправить:

```
function getEnumValue(value: TestNeverEnum): string {
   switch (value) {
      case TestNeverEnum.FIRST: return "First case";
      case TestNeverEnum.SECOND: return "Second case";
}

let returnValue: never = value; }
```

Здесь мы добавили оператор case для значения TestNeverEnum.SECOND, и поэтому последняя строка кода никогда не будет выполнена.

Использование never в качестве типа в этом случае помогает избежать ошибок логики в нашем коде. В крупных и долгоживущих проектах это может происходить довольно часто. В частности, перечисления постоянно добавляются по мере роста требований к проекту. Используя never в этих случаях, мы можем перехватывать ошибки логики, когда изменили перечисление, но не добавили правильные команды-переключатели.

Unknown

Аналогично never, TypeScript 3 представляет тип unknown. Его можно рассматривать как безопасный эквивалент типа any. Другими словами, прежде чем использовать переменную, помеченную как unknown, мы должны явно привести ее к известному типу. Давайте рассмотрим сходство между unknown и any:

```
let unknownType: unknown = "an unknown string";
console.log(`unknownType : ${unknownType}`);

unknownType = 1;
console.log(`unknownType : ${unknownType}`);
```

Здесь мы определили переменную с именем unknownType и присвоили ей строковое значение.

Затем мы записываем ее значение в консоль. Обратите внимание, что мы также явно типизировали эту переменную, чтобы она была типа unknown.

После этого мы присваиваем числовое значение 1 переменной unknownType и снова записываем ее значение в консоль. Тип этой переменной ведет себя так же, как и any, то есть мы можем переназначить ее тип на лету. Вывод этого кода выглядит так:

```
unknownType : an unknown string
unknownType : 1
```

Видно, что значение переменной unknownType изменилось со строки на число, аналогично типу any.

Однако если мы попытаемся присвоить переменную unknownType известному типу, мы увидим разницу между any и unknown:

```
let numberType: number;
numberType = unknownType;
```

Здесь у нас есть переменная с именем numberType типа number, и мы пытаемся присвоить ей значение unknownType. Это вызовет ошибку:

```
error TS2322: Type 'unknown' is not assignable to type 'number'
```

Чтобы исправить ее, нам нужно явно привести переменную unknownType к типу number:

```
numberType = <number>unknownType;
```

Обратите внимание на явное приведение в правой части оператора присваивания <number>unknownType. Этот код компилируется без ошибок.

Опять же, тип unknown рассматривается как безопасная версия типа any, так как мы вынуждены явно приводить из неизвестного типа к известному, прежде чем использовать переменную.

Object rest and spread

При работе с базовыми объектами JavaScript нам часто приходится копировать свойства одного объекта в другой или выполнять смешивание и сопоставление свойств различных объектов.

Подобно тому, как мы определили функцию как переменное число аргументов, мы можем использовать этот же синтаксис для стандартных объектов. Эта техника носит название **object rest and spread**. Рассмотрим приведенный ниже код:

```
let firstObj = { id: 1, name: "firstObj" };

let secondObj = { ...firstObj };
console.log(`secondObj : ${JSON.stringify(secondObj)}`);
```

Здесь мы начинаем с определения простого объекта JavaScript `firstObj`, у которого есть свойства `id` и `name`. Затем мы используем новый синтаксис ES7, чтобы скопировать все свойства `firstObj` в другой объект под названием `secondObj`, используя синтаксис оператора остатка (`rest`), который представляет собой многоточие перед переменной `{... firstObj}`. Чтобы проверить, что все свойства действительно скопированы, мы записываем значение переменной `secondObj` в консоль. Вывод этого кода выглядит так:

```
secondObj : {"id":1,"name":"firstObj"}
```

Здесь видно, что значения свойств `id` и `name` были скопированы из `firstObj` в `secondObj` с использованием синтаксиса оператора остатка стандарта ES7.

Мы также можем использовать этот синтаксис для объединения нескольких объектов. Это называется распространением объекта (`object spread`):

```
let nameObj = { name: "nameObj" };
let idObj = { id: 2 };

let obj3 = { ...nameObj, ...idObj };
console.log(`obj3 : ${JSON.stringify(obj3)}`);
```

Здесь у нас есть объект `nameObj`, который определяет одно свойство с именем `name`. Затем мы определяем второй объект `idObj`, который определяет одно свойство с именем `id`. После этого мы создаем `obj3` из `nameObj` и `idObj`, используя синтаксис оператора остатка `{... nameObj, ... idObj}`. Этот синтаксис означает, что мы намереваемся скопировать все свойства из `nameObj` и все свойства из `idObj` в новый объект `obj3`. Результат этого кода выглядит так:

```
obj3 : {"name":"nameObj","id":2}
```

Мы видим, что свойства обоих объектов были объединены в один, используя распространение объекта.

Приоритет распространения

При использовании распространения объекта свойства будут копироваться постепенно. Другими словами, если два объекта обладают свойством с одинаковым именем, то свойство объекта, указанное последним, будет иметь приоритет. В качестве примера рассмотрим следующий код:

```
let objPrec1 = { id: 1, name: "object prec 1" };
let objPrec2 = { id: 1001, description: "object prec 2 descripton" }
```

```
let obj4 = { ...objPrec1, ...objPrec2 };
console.log(`obj4 : ${JSON.stringify(obj4)}`);
```

Здесь мы определили два объекта, `objPrec1` и `objPrec2`. Обратите внимание, что у обоих есть свойство `id`, но у `objPrec1` есть свойство name, а у `objPrec2` есть свойство `description`. Как мы уже видели, синтаксис операторов остатка и распространения объединит свойства обоих объектов, но что он делает со свойством `id`? Это будет `1` или `1001`? Синтаксис операторов остатка и распространения будет использовать значение последнего определенного свойства. Это можно увидеть, запустив код и проверив объект `obj4`:

```
obj4 : {"id":1001,"name":"object prec 1","description":"object
prec 2 descripton"}
```

Здесь видно, что `obj4` принял значение `1001` для свойства `id`, как и ожидалось.

Использование операторов остатка и распространения с массивами

Интересно, что операторы остатка и распространения также могут использоваться с массивами. Рассмотрим следующий код:

```
let firstArray = [1, 2, 3, 4, 5];
console.log(`firstArray=${firstArray}`);

firstArray = [...firstArray, 6, 7, 8];
console.log(`firstArray=${firstArray}`);
```

Здесь мы определили имя массива `firstArray`, у которого есть пять элементов, представляющих собой числа от 1 до 5. Затем мы записываем этот массив в консоль. Обратите внимание на следующую строку из этого фрагмента кода, где мы используем синтаксис оператора остатка для добавления значений 6, 7 и 8 в существующий массив. После этого мы записываем массив `firstArray` в консоль. Вывод этого кода выглядит так:

```
firstArray=1,2,3,4,5
firstArray=1,2,3,4,5,6,7,8
```

Как видно, значения 6, 7 и 8 были добавлены к исходному массиву, используя синтаксис операторов остатка и распространения в несколько новой форме. Обратите внимание, что этот синтаксис может использоваться с массивами любого типа, а также как замена элементам массива. Рассмотрим приведенный далее код:

```
let secondArray = [
  { id: 1, name: "name1" },
```

```
    { id: 2, name: "name2" }
]
console.log(`secondArray : ${JSON.stringify(secondArray)}`);
secondArray = [
    { id: -1, name: "name-1" },
    ...secondArray,
    { id: 3, name: "name3" },

];
console.log(`secondArray : ${JSON.stringify(secondArray)}`);
```

Здесь мы начинаем с массива `secondArray`, который содержит массив сложных типов, где у каждого типа есть `id` и свойство `name`, который мы затем записываем в консоль.

Обратите внимание, однако, на третью строку этого фрагмента кода. Мы используем синтаксис операторов остатка и распространения для переопределения элементов в массиве. Этот массив теперь содержит элемент со свойством `id:-1`, затем все элементы исходного массива (с использованием синтаксиса оператора остатка ...), а потом элемент со свойством `id:3`. Вывод этого фрагмента кода выглядит так:

```
secondArray : [{"id":1,"name":"name1"}, {"id":2,"name":"name2"}]
secondArray : [{"id":-1,"name":"name-1"}, {"id":1,"name":"name1"},
{"id":2,"name":"name2"}, {"id":3,"name":"name3"}]
```

Как видно из выходных данных, массив `secondArray` начинался только с двух элементов, а затем, используя синтаксис операторов остатка и распространения, мы вставили по элементу в начале и в конце массива.

Кортежи

Кортежи – это метод определения типа, у которого есть конечное число неназванных свойств.

У каждого свойства есть связанный тип. При использовании кортежа должно быть указано каждое из этих свойств. Лучше всего это можно объяснить в следующем примере:

```
let tupleType: [string, boolean];
tupleType = ["test", false];
tupleType = ["test"];
```

Здесь мы определили переменную с именем `tupleType`, тип которой определен как массив типов, первый из которых является строкой (`string`), а второй – логическим типом данных (`boolean`). Затем мы присваиваем значение переменной `tupleType` и используем синтаксис массива для установки первого свойства как

строки "test", а второго – как логического значения false. Обратите внимание, что в последней строке этого фрагмента мы пытаемся присвоить значение переменной tupleType, которая имеет только свойство строки. Эта строка выдаст ошибку:

```
error TS2322: Type '[string]' is not assignable to type
'[string, boolean]'
```

Эта ошибка показывает, что для использования кортежа должно быть установлено каждое из свойств кортежа и что он рассматривается компилятором как тип с двумя свойствами.

Кортежи обычно используются, когда нам нужно временно связать два обычно не связанных свойства.

Деконструкция кортежей

Поскольку кортежи используют синтаксис массива, их можно деконструировать или дизассемблировать двумя способами. Первый с помощью простого синтаксиса массива выглядит так:

```
console.log(`tupleType[0] : ${tupleType[0]}`);
console.log(`tupleType[1] : ${tupleType[1]}`);
```

Здесь мы просто записываем каждое свойство переменной tupleType в консоль, обращаясь к индексу в массиве, то есть tupleType[0] и typleType[1]. Вывод этого кода будет таким:

```
tupleType[0] : test
tupleType[1] : false
```

Итак, мы создали кортеж со строкой и логическим значением и деконструировали его с помощью синтаксиса массива. Обратите внимание, что поскольку мы используем синтаксис массива, то можем запросить третье свойство этого кортежа:

```
console.log(`tupleType[2] : ${tupleType[2]}`);
```

Поскольку у нашего кортежа нет третьего свойства, typleType[2] будет неопределенным, как видно из вывода этой строки кода:

```
tupleType[2] : undefined
```

Это явно далеко от идеала.

Лучший способ деконструировать кортеж – использовать синтаксис массива для создания соответствующего кортежа в левой части присваивания:

```
let [t1, t2] = tupleType;
```

```
console.log(`t1: ${t1}`);
console.log(`t2: ${t2}`);
```

Здесь мы определяем массив из двух элементов с именами `t1` и `t2` и присваиваем этому массиву значение кортежа. Затем мы записываем `t1` и `t2` в консоль. Вывод этого кода выглядит так:

```
t1: test
t2: false
```

Данный метод деконструкции кортежа предпочтителен по простой причине. Мы не можем определить массив элементов, который превышает количество свойств в кортеже. Следовательно, приведенный ниже код работать не будет:

```
let [et1, et2, et3] = tupleType;
```

Здесь мы пытаемся деконструировать наш кортеж из двух свойств в кортеж из трех свойств. Компилятор в этом случае выдаст ошибку:

```
error TS2493: Tuple type '[string, boolean]' with length '2'
cannot be assigned to tuple with length '3'
```

Необязательные элементы кортежа

Подобно сигнатурам функций, у нас также могут быть необязательные элементы кортежа. Это достигается с помощь символа ? в определении кортежа:

```
let optionalTuple: [string, boolean?];
optionalTuple = ["test2", true];
console.log(`optionalTuple : ${optionalTuple}`);
optionalTuple = ["test"];
console.log(`optionalTuple : ${optionalTuple}`);
```

Здесь у нас есть определенная переменная с именем `optionTuple` с обязательным свойством `string` и необязательным свойством `boolean`. Затем мы присваиваем ей значение `["test2", true]` и записываем в консоль. После этого мы присваиваем значение `["test"]` тому же кортежу и записываем значение в консоль. Так как второе свойство `optionTuple`, по сути, необязательно, данный код будет скомпилирован чисто и даст, как и ожидалось, следующие результаты:

```
optionalTuple : test2,true
optionalTuple : test
```

Кортежи и синтаксис оператора остатка

Кортежи также могут использовать синтаксис оператора остатка как в определениях функций, так и в объектах `rest` и `spread`.

В качестве примера использования кортежей и оператора остатка в определении функции рассмотрим следующий код:

```
function useTupleAsRest(...args: [number, string, boolean]) {
   let [arg1, arg2, arg3] = args;
   console.log(`arg1: ${arg1}`);
   console.log(`arg2: ${arg2}`);
   console.log(`arg3: ${arg3}`);
}
useTupleAsRest(1, "stringValue", false);
```

Здесь мы определили функцию с именем `useTupleAsRest`, которая использует синтаксис оператора остатка, но при этом определяет параметр args как кортеж. Это позволяет нам разделить кортеж на три переменные `arg1`, `arg2` и `arg3` в первой строке данной функции. После чего мы записываем все три значения в консоль. Когда мы вызываем эту функцию, то должны предоставить три значения, которые соответствуют типам кортежей. Вывод этого кода выглядит так:

```
arg1: 1
arg2: stringValue
arg3: false
```

Обратите внимание, что поскольку параметр остатка теперь является кортежем, мы получим ошибки компиляции, если попытаемся вызвать эту функцию с чем-либо, кроме комбинации числа, строки или логического типа данных.

Второй способ использования синтаксиса оператора остатка с кортежами состоит в их определении.

Рассмотрим следующий код:

```
type RestTupleType = [number, ...string[]];
let restTuple: RestTupleType = [1, "string1", "string2", "string3"];
```

Здесь мы определили тип с именем `RestTupleType`, который представляет собой кортеж с первым свойством числа, а затем с переменным числом строк (используя синтаксис оператора остатка). Затем мы создаем переменную `restTuple` с номером 1 и строками `"string1"`, `"string2"` и `"string3"` в качестве свойств.

Bigint

В состав предложений по стандарту ECMAScript недавно вошла поддержка обработки действительно больших чисел. Большинство языков программирования уже изначально поддерживает 64-битные числа, но JavaScript отстает в этом отношении и в настоящее время поддерживает только числа с точностью 53 бита. Причины этого конкретного ограничения достаточно детализированы и сво-

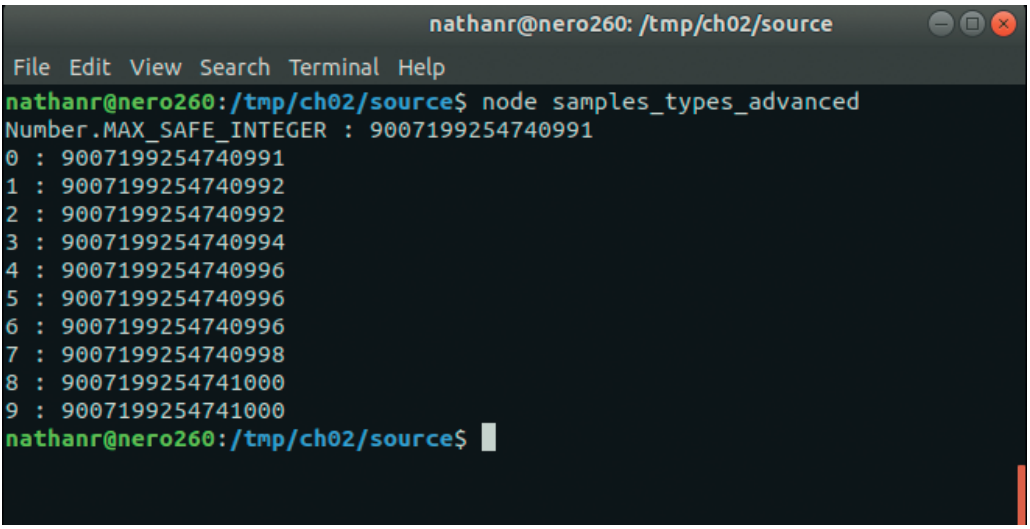
дятся к внутреннему представлению, которое JavaScript использует для хранения чисел в памяти. Любое числовое представление в памяти должно учитывать как знак, положительный или отрицательный, так и точность числа, или, другими словами, количество десятичных знаков. В JavaScript наибольшее число, которое можно использовать при использовании точности 53 бита, составляет `9,007,199,254,740,991`. С точки зрения непрофессионала это: девять квадриллионов, семь триллионов, сто девяносто девять миллиардов, двести пятьдесят четыре миллиона, семьсот сорок тысяч, девятьсот девяносто один. Это очень, очень, очень большое число.

Хотя нам может и не потребоваться работать с девятью квадриллионами возможных различных значений в нашем коде, эти типы чисел действительно возникают при определенных обстоятельствах. Достаточно лишь взглянуть на современные процедуры криптографии, чтобы найти примеры. Например, если ваше приложение работает с банком, банк может сгенерировать какой-либо числовой маркер, который зашифрован с помощью расширенной процедуры криптографии, чтобы представить уникальный идентификатор транзакции определенного платежа. Чем больше это число, тем сложнее его расшифровать и тем безопаснее система. Поэтому получить 64-битное число в качестве уникального идентификатора вполне возможно.

Давайте посмотрим на ограничения текущего типа number в JavaScript с помощью приведенного ниже кода:

```
console.log(`Number.MAX_SAFE_INTEGER : ${Number.MAX_SAFE_INTEGER}`);

let highest53bitNumber = 9_007_199_254_740_991;

for (let i = 0; i < 10; i++) {
   console.log(`${i} : ${highest53bitNumber + i}`);
}
```

Здесь мы начинаем с записи значения константы `Number.MAX_SAFE_INTEGER` в консоль. Эта константа будет возвращать число, которое является максимальным значением целого числа, которое поддерживает JavaScript. Затем мы определяем переменную с именем `high53bitNumber` и устанавливаем для нее это максимальное значение. После этого код выполняет простой цикл `for`, который добавляет к этому числу числа от `0` до 9 и записывает результаты в консоль. Вывод этого кода выглядит так:



Здесь мы видим довольно странные результаты. Добавление значения 2 в переменную `high53bitNumber` или добавление значения 1 неожиданно приводит к одному и тому же результату.

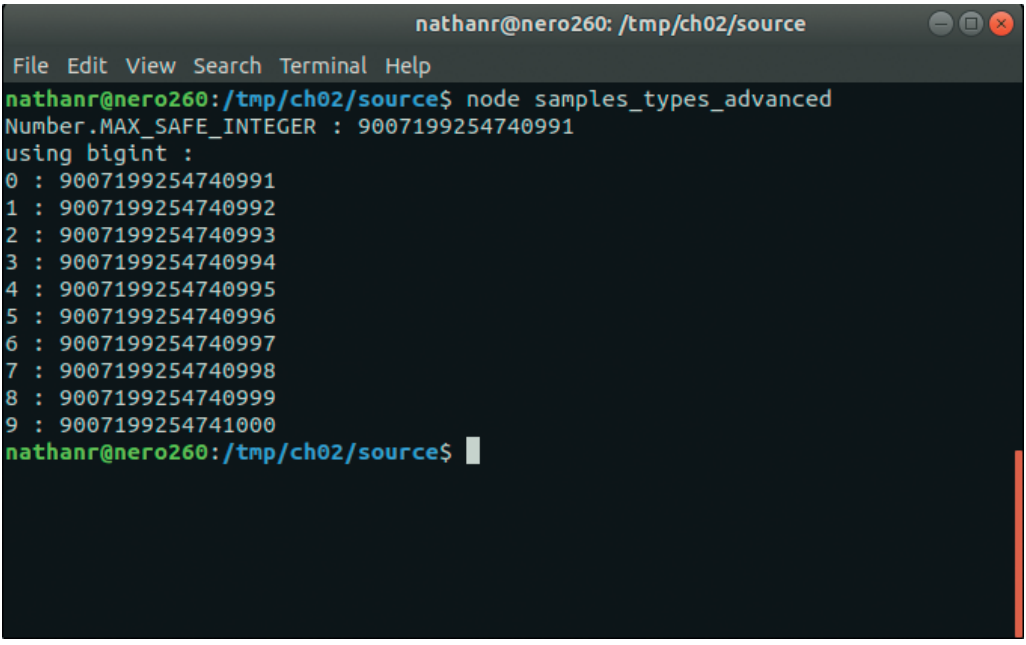
Добавление значения 4, 5 или 6 дает тот же результат. Здесь мы видим результаты попыток выполнить простую арифметику для чисел, выходящих за пределы `Number.MAX_SAFE_INTEGER`, поддерживаемые в JavaScript. Поскольку JavaScript не поддерживает числа, превышающие это ограничение, результаты являются неопределенными.

В последних версиях стандарта ECMAScript реализован новый базовый тип под названием `bigint` для обработки такого рода действительно больших чисел. Давайте посмотрим на тот же цикл из предыдущего фрагмента кода с использованием `bigint`:

```
console.log(`using bigint :`);
let bigIntNumber: bigint = 9_007_199_254_740_991n;
for (let i = 0; i < 10; i++) {
   console.log(`${i} : ${bigIntNumber + BigInt(i)}`);
}
```

Здесь мы определили переменную `bigIntNumber` и указали ее тип: `bigint`. Тип `bigint` является дополнением к базовым типам `string`, `number` и `boolean` и рассматривается аналогичным образом. Это означает, что мы не можем назначить тип `number` типу `bigint` так же, как не можем назначить тип `string` типу `number`. Обратите внимание на определение значения `bigint` в данном случае. Мы добавили букву `n` к числовому значению, чтобы компилятор распознал, что мы определяем значение `bigint`.

Наш цикл аналогичен предыдущему фрагменту кода в том, что он перебирает значения от `0` до `9`, добавляет это значение в нашу переменную `bigIntNumber`, а затем записывает результаты в консоль. Обратите внимание, однако, что нам нужно создать значение `bigint` из переменной `i`, которая на самом деле типа `number`. Это достигается путем вызова статической функции `BigInt` и передачи нашего числового типа в качестве аргумента. Другими словами, `BigInt(i)` преобразует переменную `i` типа `number` в тип `bigint`. Запуск этого кода теперь дает следующие результаты:



Здесь видно, что, используя новый собственный тип `bigint`, мы можем выполнять арифметические вычисления над числами, которые превышают допустимую двойную точность (53 бита).

Реализация `bigint` не была поддержана компилятором TypeScript для ES5 или ES3, или любой другой версии ниже Esnext. Это означает, что мы должны установить в качестве цели esnext в нашем файле `tsconfig.json`, чтобы успешно скомпилировать код, использующий тип `bigint`. Еще одно предостережение относительно данного вопроса состоит в том, что только два механизма времени исполнения JavaScript реализовали поддержку типа `bigint` на момент написания этой главы, поэтому он действительно является передовым. Вам потребуется запустить последнюю версию Chrome либо Node версии 11 и выше, чтобы использовать тип `bigint`.

Резюме

В этой главе мы рассмотрели основные типы TypeScript, переменные и методы функций. Мы увидели, как TypeScript использует синтаксический сахар поверх обычного кода JavaScript для обеспечения сильно типизированных переменных и сигнатур функций. Мы также увидели, как TypeScript использует утиную типизацию и явное приведение, а затем обсудили функции TypeScript, сигнатуры функций и переопределения. Мы завершили главу обсуждением расширенных типов, включая охранников типов, операторы остатка и распределения, кортежи и `Bigint`.

В следующей главе мы будем опираться на эти знания и увидим, как TypeScript распространяет эти сильно типизированные правила на концепции ООП, такие как интерфейсы, классы и наследование.

Глава 3

Интерфейсы, классы и наследование

Мы уже видели, как TypeScript использует базовые типы, выводимые типы, сигнатуры функций и кортежи, чтобы привнести опыт сильно типизированной разработки в JavaScript. Эта сильно типизированная парадигма также включает в себя свойства ООП, аналогичные другим языкам, такие как интерфейсы, классы и наследование. Там, где у TypeScript всегда были языковые конструкции для интерфейсов и классов, эти свойства были только недавно ратифицированы стандартом ECMAScript 6. Это означает, что для использования этих свойств ООП в JavaScript наша целевая среда должна поддерживать стандарт ES6.

Однако TypeScript позаботится о создании ES3- или ES5-совместимого JavaScript, то есть независимо от того, на какую среду исполнения JavaScript мы нацелены, мы можем использовать все совершенство объектно-ориентированного TypeScript для проектирования и создания своих приложений. В данной главе мы рассмотрим эти концепции ООП, их использование в TypeScript и какие преимущества они привносят в опыт разработки на JavaScript.

Эта глава разбита на два основных раздела. Первый раздел предназначен для читателей, которые используют TypeScript впервые, и рассказывает об интерфейсах, классах и наследовании с нуля. Второй раздел основывается на этих знаниях и показывает, как создавать и использовать классы, интерфейсы и наследование путем создания образца шаблона проектирования Factory.

Если у вас есть опыт работы с TypeScript, вы активно используете интерфейсы и классы и понимаете наследование, тогда непременно просмотрите эту главу. Возможно, вас больше заинтересуют некоторые ключевые слова, которые предоставляет TypeScript для сравнения структуры классов, такие как слабые типы, оператор `instanceof` или оператор `in`. В последних разделах главы обсуждается шаблон проектирования Factory и абстрактные классы.

В этой главе будут рассмотрены следующие темы:

- интерфейсы;
- слабые типы;
- оператор `in`;

- классы;
- конструкторы классов;
- модификаторы классов;
- статические функции и свойства;
- наследование;
- абстрактные классы;
- замыкания JavaScript;
- instanceof и охранники типов;
- шаблон проектирования Factory.

Интерфейсы

Интерфейс предоставляет механизм для определения того, какие свойства и методы должен реализовывать объект, и, следовательно, является способом определения пользовательского типа. Мы уже исследовали синтаксис TypeScript для сильной типизации переменной в один из основных типов, таких как строка или число. Используя этот синтаксис, мы также можем строго типизировать переменную как пользовательский тип, или, более правильно, тип интерфейса. Это означает, что у переменной должны быть те же свойства, что описаны в интерфейсе. Если объект придерживается интерфейса, говорят, что объект реализует интерфейс. Интерфейсы определяются с помощью ключевого слова `interface`.

Чтобы проиллюстрировать концепцию интерфейсов, рассмотрим следующий код:

```
interface IComplexType {
   id: number;
   name: string; }
```

Мы начнем с интерфейса с именем `IComplexType`, у которого есть `id` и свойство `name`. Свойство `id` сильно типизировано и имеет тип `number`, а свойство `name` имеет тип `string`. Данное определение интерфейса может быть применено к переменной:

```
let complexType : IComplexType;
complexType = { id: 1, name : "test" };
```

В данном случае мы определили переменную `complexType` и строго типизировали ее как тип `IComplexType`. Затем мы создаем экземпляр объекта и присваиваем значения свойствам объекта. Обратите внимание, что интерфейс `IComplexType` определяет и `id` и свойство `name`, и как таковые они оба должны присутствовать. Если мы попытаемся создать экземпляр объекта без этих свойств:

```
let incompleteType : IComplexType;
incompleteType = { id : 1};
```

TypeScript выдаст ошибку:

```
error TS2322: Type '{ id: number; }' is not assignable
to type 'IComplexType'.
Property 'name' is missing in type '{ id: number; }'.
```

Необязательные свойства

Определения интерфейса могут также включать необязательные свойства, аналогично тому, как функции могут иметь необязательные свойства. Рассмотрим следующее определение интерфейса:

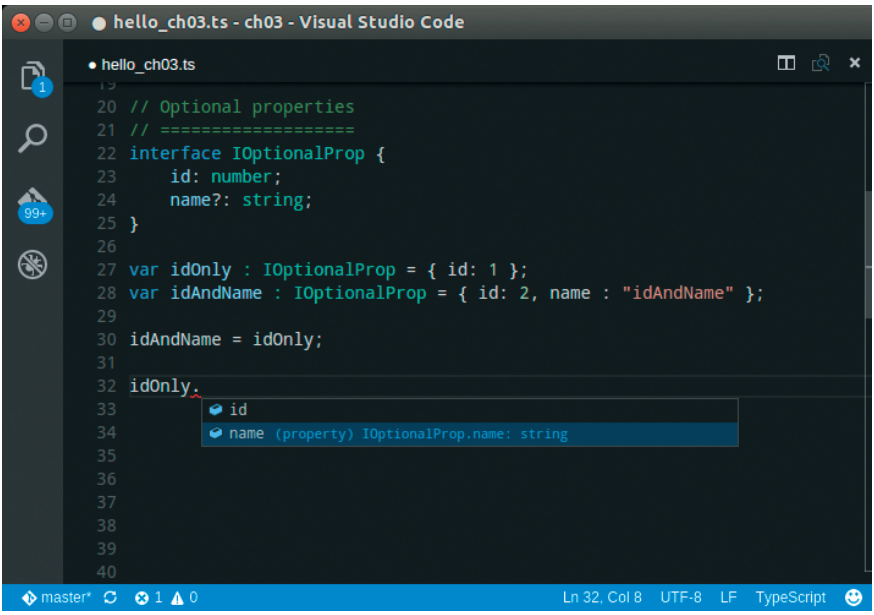
```
interface IOptionalProp {
   id: number;
   name?: string;
}
```

Здесь мы определили интерфейс с именем `IOptionalProp`, у которого есть свойство `id` типа `number` и необязательное свойство `name` типа `string`. Обратите внимание, что синтаксис необязательных свойств аналогичен тому, что мы видели в случае с необязательными параметрами при определении функций. Другими словами, символ ? после свойства `name` используется для указания того, что это свойство является необязательным. Следовательно, мы можем использовать это определение интерфейса следующим образом:

```
let idOnly : IOptionalProp = { id: 1 };
let idAndName : IOptionalProp = { id: 2, name : "idAndName" };
idAndName = idOnly;
```

Здесь у нас есть две переменные, каждая из которых реализует интерфейс `IOptionalProp`. Первая переменная `idOnly` просто указывает свойство `id`. Это допустимый код TypeScript, поскольку мы пометили свойство `name` интерфейса `IOptionalProp` как необязательное. Вторая переменная называется `idAndName` и определяет как свойство `id`, так и `nam`. Обратите внимание на последнюю строку этого фрагмента кода. Поскольку обе переменные реализуют интерфейс `IOptionalProp`, мы можем назначить их друг другу. Без использования определения интерфейса с необязательными свойствами этот код обычно вызывал бы ошибку.

Если вы используете редактор, который поддерживает TypeScript, у вас должно появиться автозаполнение кода или подсказки IntelliSense, возникающие при работе с интерфейсами, потому что ваш редактор использует языковой сервис TypeScript, чтобы автоматически определить тип, с которым вы работаете, и какие свойства и функции доступны:



Здесь видно, что редактор VSCode использует IntelliSense, чтобы автоматически показать нам, какие свойства доступны для переменной `idOnly`. Поскольку она определена как реализующая интерфейс `IOptionalProp`, у нее есть два доступных свойства, которые отображаются в редакторе.

Компиляция интерфейса

Интерфейсы являются свойством языка времени компиляции TypeScript, и компилятор не генерирует никакого кода JavaScript из интерфейсов, которые вы включаете в свои проекты TypeScript. Интерфейсы используются компилятором только для проверки типов в ходе шага компиляции.

В этой книге мы будем придерживаться простого соглашения об именовании интерфейсов, а именно ставить перед именем интерфейса букву `I`. Использование такой схемы именования помогает при работе с большими проектами, где код распространяется на несколько файлов. Если вы видите в своем коде что-либо с префиксом `I`, то сразу же можете выделить это как интерфейс. Однако для своих интерфейсов можно использовать любой стандарт именования.

Слабые типы

Когда интерфейс содержит только необязательные свойства, он считается слабым типом. Деругими словами, если мы создаем объект, который реализует

интерфейс слабого типа, можем ли мы действительно сказать, что объект реализует интерфейс? Давайте рассмотрим приведенный ниже код, чтобы выяснить это:

```
interface IWeakType {
   id?: number,
   name?: string
}
let weakTypeNoOverlap: IWeakType;
weakTypeNoOverlap = { description: "my description" };
```

Здесь мы определили интерфейс с именем `IWeakType`, который содержит два свойства, `id` и `name`. Поскольку оба этих свойства являются необязательными, данный интерфейс считается слабым типом. Затем мы определяем переменную `weakTypeNoOverlap` и указываем этот объект как тип `IWeakType`. В последней строке этого кода мы присваиваем значение данной переменной, которая имеет свойство `description`. Обратите внимание, что нет никакого перекрытия между свойствами, которые мы присвоили переменной, и интерфейсом, который она должна реализовывать.

В этом случае компилятор TypeScript выдаст следующую ошибку:

```
error TS2322: Type '{ description: string; }' is not assignable
to type 'IWeakType'.
Object literal may only specify known properties, and
'description' does not exist in type 'IWeakType'.
```

Здесь видно, что компилятор обнаруживает, что переменная `weakTypeNoOverlap` должна реализовывать интерфейс `IWeakType` или, по крайней мере, некоторую его часть. Так как между свойствами интерфейса `IWeakType` и присваиванием переменной нет перекрытия, присваивание приведет к ошибке.

Вывод типов с помощью оператора in

TypeScript также позволяет проверять объект на наличие свойства при помощи оператора `in`. Представьте, что у нас есть следующие интерфейсы:

```
interface IHasIdAndNameProperty {
   id: number;
   name: string;
}

interface IHasDescAndValueProperty {
   description: string;
   value: number;
}
```

Здесь у нас есть интерфейс `IHasIdAndNameProperty`, который определяет `id` и свойство `name`. У нас также есть интерфейс `IHasDescAndValueProperty`, который определяет свойства `description` и `value`. Теперь мы можем написать функцию, которая может работать с обоими интерфейсами:

```
function printNameOrDescription(
   value: IHasIdAndNameProperty | IHasDescAndValueProperty) {
      if ('id' in value) {
         console.log(`found id ! | name : ${value.name}`);
      }
      if ('value' in value) {
         console.log(`found value ! : description :
         ${value.description}`);
      }
   }
```

Здесь у нас есть функция с именем `printNameOrDescription` с единственным аргументом `value`. Тип аргумента `value` представляет собой объединение интерфейсов `IHasIdAndNameProperty` и `IHasDescAndValueProperty`. Обратите внимание, что в свойствах нет перекрытия между интерфейсами, так как определить, какой интерфейс был передан? В этом случае мы используем охранника типов, который использует оператор `in`:

```
if ('id' in value)
```

Наш первый охранник типов тестирует передаваемый аргумент и выполняет проверку на предмет наличия свойства с именем `id`. Если это так, то мы знаем, что переданный тип был `IHasIdAndNameProperty`, и поэтому можем вывести значение свойства `name` в консоль. Аналогично, второй охранник проверяет наличие свойства `value`:

```
if ('value' in value)
```

Если свойство `value` найдено, то мы знаем, что природа аргумента `value` имеет тип `IHasDescAndNameProperty`, и можем записать значение свойства `description` в консоль.

Следовательно, TypeScript позволяет нам генерировать охранников типов для интерфейсов, используя оператор `in`, чтобы проверить, обладает ли интерфейс определенным свойством.

Классы

Класс – это определение объекта, какие данные он содержит и какие операции он может выполнять. Классы и интерфейсы являются краеугольным камнем прин-

ципов объектно-ориентированного программирования. Давайте посмотрим на простое определение класса:

```
class SimpleClass {
   id: number;
   print() : void {
      console.log(`SimpleClass.print() called`);
   }
}
```

Здесь мы использовали ключевое слово `class`, чтобы определить класс с именем `SimpleClass`, и мы определили его, чтобы получить свойство с именем `id` и функцию `print`. Свойство `id` было определено как свойство типа `number`. Функция `print` просто записывает сообщение в консоль. Обратите внимание, что все, что мы здесь сделали, – это определили, какие данные может хранить этот класс и что он может делать. Однако если мы попытаемся скомпилировать этот код, появится ошибка:

```
error TS2564: Property 'id' has no initializer and is not
definitely assigned in the constructor
```

Эта ошибка указывает на то, что свойство `id` без явной установки по-прежнему может быть не определено (`undefined`). Если мы попытаемся использовать свойство `id`, а оно не было установлено в числовое значение, то нас может ожидать сюрприз, если оно не определено. Есть несколько способов справиться с этой ошибкой, но на данный момент мы можем просто сделать свойство `id` объединением типов:

```
id: number | undefined;
```

Добавляя сюда объединение типов, мы принимаем сознательное решение о том, что свойство `id` может быть неопределенным. Это пример того, как компилятор TypeScript может обнаруживать ошибки в нашем коде, о которых мы, возможно, вначале и не думали.

Чтобы использовать это определение класса, нам нужно создать экземпляр класса:

```
let mySimpleClass = new SimpleClass();
mySimpleClass.print();
```

Здесь мы определяем переменную `mySimpleClass` для хранения экземпляра класса `SimpleClass`, а затем создаем этот класс, используя ключевое слово `new`. После создания экземпляра этого класса мы можем вызвать функцию `print` для нашего экземпляра класса. Вывод этого кода будет таким:

```
SimpleClass.print() called
```

Свойства класса

Класс является определением объекта и как таковой включает в себя данные, которые он может хранить, в виде свойств класса. Эти свойства могут быть доступны пользователям данного класса, если мы того пожелаем, или могут быть помечены как недоступные. Независимо от того, каковы их правила доступности, код в классе должен будет получить доступ к этим свойствам. Чтобы получить доступ к свойствам класса из самого класса, нужно использовать ключевое слово `this`. В качестве примера давайте обновим определение класса `SimpleClass` и выведем значение свойства `id` в функции `print`:

```
class SimpleClass {
   id: number | undefined;
   print() : void {
      console.log(`SimpleClass has id : ${this.id}`);
   }
}
```

Здесь мы изменили функцию print, чтобы обратиться к свойству `id` экземпляра класса в строке шаблона, `${this.id}`. Всякий раз, когда мы находимся внутри экземпляра класса, мы должны использовать ключевое слово `this` для доступа к любому свойству или функции, доступным в определении класса. Следовательно, ключевое слово `this` указывает компилятору, что мы обращаемся к свойству класса или его функции.

Как только мы создали экземпляр класса, мы можем установить свойство `id`, а затем вызвать обновленную функцию `print`:

```
let mySimpleClass = new SimpleClass();
mySimpleClass.id = 1001;
mySimpleClass.print();
```

Вывод этого кода будет таким:

```
SimpleClass has id : 1001
```

Здесь мы установили свойство `id` переменной `mySimpleClass` и увидели, как функция `print` может получить доступ к свойству `id` из определения самого класса, чтобы вывести значение в консоль.

Реализация интерфейсов

Прежде чем мы продолжим изучение классов, давайте посмотрим на связь между классами и интерфейсами. Класс – это определение объекта, включая его свойства и функции. Интерфейс – это определение пользовательского типа, в том числе его свойств и функций. Единственное реальное отличие состоит в том, что классы должны реализовывать функции и свойства, тогда как интерфейсы только опи-

сывают их. Это позволяет использовать интерфейсы для описания некоторого общего поведения группы классов, а затем писать код, который будет работать с любым из этих классов. Рассмотрим следующие определения классов:

```
class ClassA {
   print() {console.log('ClassA.print()')};
}

class ClassB {
   print() {console.log('ClassB.print()')};
}
```

Здесь у нас есть определения двух классов, `ClassA` и `ClassB`. У обоих есть просто функция `print`. Предположим, что мы хотели написать некоторый код, которому на самом деле все равно, какой тип класса мы использовали. Все, что его заботило, – это наличие у класса функции `print`. Вместо того чтобы писать сложный класс, который должен был иметь дело с экземплярами `ClassA` и `ClassA`, мы можем легко создать интерфейс, описывающий необходимое нам поведение:

```
interface IPrint {
   print() : void;
}
function printClass( a : IPrint ) {
   a.print();
}
```

Здесь мы создали интерфейс с именем `IPrint` для описания атрибутов объекта, который нам нужен в функции `printClass`. У этого интерфейса есть единственная функция `print`. Поэтому любая переменная, переданная в качестве аргумента в функцию `printClass`, сама должна иметь функцию с именем `print`.

Теперь мы можем изменить наши определения классов, чтобы они оба могли использоваться функцией `printClass`:

```
class ClassA implements IPrint {
   print() {console.log('ClassA.print()')};
}

class ClassB implements IPrint {
   print() {console.log('ClassB.print()')};
}
```

В наших определениях классов теперь используется ключевое слово `implements` для реализации интерфейса `IPrint`. Это позволяет нам использовать оба класса в функции `printClass`:

```
let classA = new ClassA();
let classB = new ClassB();
```

```
printClass(classA);
printClass(classB);
```

Здесь мы создаем экземпляры `ClassA` и `ClassB`, а затем вызываем ту же самую функцию `printClass` с обоими экземплярами. Поскольку эта функция написана для принятия любого объекта, который реализует интерфейс `IPrint`, она будет корректно работать с обоими типами классов.

Следовательно, интерфейсы являются способом описания поведения класса. Их также можно рассматривать как вид контракта, который классы должны выполнять, если ожидается, что они будут обеспечивать определенное поведение.

Конструкторы классов

Классы могут принимать параметры при их первоначальном построении. Это позволяет объединить создание класса и установку его параметров в одну строку кода. Рассмотрим следующее определение класса:

```
class ClassWithConstructor {
   id: number;
   name: string;
   constructor(_id: number, _name: string) {
      this.id = _id;
      this.name = _name; }
}
```

Здесь мы определили класс с именем `ClassWithConstructor`, у которого есть два свойства, свойство `id` типа `number` и свойство `name` типа `string`. У него также есть функция-конструктор, которая принимает два параметра. Функция-конструктор присваивает значение аргумента `_id` свойству класса `id` и значение аргумента `_name` свойству класса `name`. Обратите внимание, что поскольку мы инициализируем свойства `id` и `name` в конструкторе класса, то знаем, что любой созданный класс установит значение для этих свойств. Поэтому нам не нужно использовать тип `unspecified` в этом случае.

Затем мы можем построить экземпляр этого класса следующим образом:

```
var classWithConstructor = new ClassWithConstructor(1, "name");
console.log(`classWithConstructor =
     ${JSON.stringify(classWithConstructor)}`);
```

Первая строка этого кода создает экземпляр класса `ClassWithConstructor`, используя функцию-конструктор. Затем мы просто записываем версию JSON этого класса в консоль. Вывод данного кода выглядит так:

```
classWithConstructor = {"id":1,"name":"name"}
```

Функции класса

Все функции в классе придерживаются синтаксиса и правил, которые мы рассматривали в предыдущей главе, посвященной функциям. Чтобы освежить эту информацию, напомним, что все функции класса могут:

- быть строго типизированными;
- использовать ключевое слово any, чтобы ослабить сильную типизацию;
- иметь необязательные параметры;
- иметь параметры по умолчанию;
- использовать массивы аргументов или синтаксис оставшихся параметров;
- разрешать функции обратного вызова и указывать сигнатуру этих функций;
- разрешать перегрузки функций.

В качестве примера для каждого из этих правил давайте рассмотрим класс, у которого несколько различных сигнатур функций, а затем мы подробно обсудим каждую из них:

```
class ComplexType implements IComplexType {
   id: number;
   name: string;
   constructor(idArg: number, nameArg: string);
   constructor(idArg: string, nameArg: string);
   constructor(idArg: any, nameArg: any) {
      this.id = idArg;
      this.name = nameArg;
   }
   print(): string {
      return "id:" + this.id + " name:" + this.name;
   }
   usingTheAnyKeyword(arg1: any): any {
      this.id = arg1;
   }
   usingOptionalParameters(optionalArg1?: number) {
      if (optionalArg1) {
         this.id = optionalArg1;
      }
   }

   usingDefaultParameters(defaultArg1: number = 0) {
      this.id = defaultArg1;
   }
   usingRestSyntax(...argArray: number []) {
      if (argArray.length > 0) {
         this.id = argArray[0];
      }
   }
```

```
    usingFunctionCallbacks( callback: (id: number) => string ) {
      callback(this.id);
    }
  }
```

Первое, что нужно отметить, – это функция-конструктор. В нашем определении класса используется функция, переопределяющая функцию-конструктор, что позволяет создавать класс с использованием числа и строки или двух строк. Приведенный ниже код показывает, как мы будем использовать каждое из этих определений конструктора:

```
let ct_1 = new ComplexType(1, "ct_1");
let ct_2 = new ComplexType("abc", "ct_2");
let ct_3 = new ComplexType(true, "test");
```

Переменная `ct_1` использует вариант `number`, `string` функции-конструктора, а переменная `ct_2` – вариант `strin`, `strin`. Переменная `ct_3` выдаст ошибку компиляции, так как мы не разрешаем конструктору применять вариант `boolean`, `boolean`. Однако вы можете поспорить, что последняя функция-конструктор задает вариант `any`, `any`, а это должно разрешить использование `boolean`, `boolean`. Просто помните, что перегрузки конструктора следуют тем же правилам, что и перегрузки функций, которые мы обсуждали в `главе 2` *«Типы, переменные и методы функций»*, поэтому это запрещено.

Однако мы должны быть осторожны при использовании переопределений конструктора. Давайте подробнее рассмотрим, что происходит, когда мы вызываем вариант `strin`, `strin`:

```
let ct_2 = new ComplexType("abc", "ct_2");
ct_2.print();
```

В функции-конструкторе мы присваиваем значение аргумента `idArg` свойству `id` в классе:

```
class ComplexType implements IComplexType {
  id: number;
  name: string;
  constructor(idArg: number, nameArg: string);
  constructor(idArg: string, nameArg: string);

  constructor(idArg: any, nameArg: any) {
    this.id = idArg;
    // осторожно - присваивание строки типу number;
    this.name = nameArg;
  }
}
```

Несмотря на то что мы определили тип свойства `id` как `number`, когда мы вызываем функцию-конструктор со строкой (потому что у нас есть переопределение конструктора), свойство `id` на самом деле будет `abc`, что явно не является типом number. TypeScript в этом случае не выдаст ошибку и не будет автоматически пытаться преобразовать значение `"abc"` в число. В тех случаях, когда требуется данный тип функциональности, необходимо использовать охранников типов для обеспечения их безопасности:

```
constructor(idArg: any, nameArg: any) {
   if (typeof idArg === "number") {
      this.id = idArg; }
   this.name = nameArg;
}
```

Здесь мы использовали охранника типов, чтобы гарантировать, что свойство `id` (тип `number`) присваивается только в том случае, если параметр `idArg` фактически является числом.

Давайте теперь посмотрим на остальные определения функций, которые мы определили для класса, начиная с функции `usingTheAnyKeyword`:

```
ct_1.usingTheAnyKeyword(true);
ct_1.usingTheAnyKeyword({ id: 1, name: "string"});
```

Первый вызов в этом примере использует логическое значение для вызова функции `usingTheAnyKeyword`, а второй использует произвольный объект. Оба этих вызова функций валидны, так как параметр `arg1` определен с типом any.

Далее идет функция `usingOptionalParameters`:

```
ct_1.usingOptionalParameters(1);
ct_1.usingOptionalParameters();
```

Здесь мы вызываем функцию `usingOptionalParameters` сначала с одним аргументом, а затем без каких-либо аргументов. Опять же, эти вызовы валидны, поскольку аргумент `optionArg1` помечен как необязательный.

А теперь функция `usingDefaultParameters`:

```
ct_1.usingRestSyntax(1,2,3);
ct_2.usingRestSyntax(1,2,3,4,5);
```

Оба этих вызова функции `usingDefaultParameters` являются валидными. Первый вызов переопределит значение по умолчанию `0`, а второй вызов – без аргумента – будет использовать значение по умолчанию `0`.

Далее следует функция `usingRestSyntax`:

```
ct_1.usingRestSyntax(1,2,3);
ct_2.usingRestSyntax(1,2,3,4,5);
```

Наша функция `usingRestSyntax` может быть вызвана с любым количеством аргументов, так как мы используем синтаксис оставшихся параметров для хранения этих аргументов в массиве. Оба эти вызова валидны.

Наконец, давайте посмотрим на функцию `usingFunctionCallbacks`:

```
function myCallbackFunction(id: number): string {
   return id.toString();
}
ct_1.usingFunctionCallbacks(myCallbackFunction);
```

Здесь мы определили функцию `myCallbackFunction`, которая соответствует сигнатуре функции обратного вызова, предусмотренной функцией `usingFunctionCallbacks`. Это позволяет передавать `myCallbackFunction` в качестве параметра в функцию `usingFunctionCallbacks`.

Обратите внимание, что если у вас возникнут трудности с пониманием различных сигнатур функций, просмотрите соответствующие разделы в `главе 2` *«Типы, переменные и методы функций»*, относящиеся к функциям, где каждое из этих понятий объясняется подробно.

Определения функций интерфейса

Интерфейсы, подобно классам, следуют одним и тем же правилам при работе с функциями. Чтобы обновить определение интерфейса `IComplexType`, дабы оно соответствовало определению класса `ComplexType`, нам нужно написать определение для каждой новой функции:

```
interface IComplexType {
   id: number;
   name: string;
   print(): string;
   usingTheAnyKeyword(arg1: any): any;
   usingOptionalParameters(optionalArg1?: number) : void;
   usingDefaultParameters(defaultArg1?: number) : void;
   usingRestSyntax(...argArray: number []) : void;
   usingFunctionCallbacks(callback: (id: number) => string);
}
```

Данное определение интерфейса включает в себя свойства `id` и `name` и функцию `print`. Затем у нас есть сигнатура функции для функции `usingTheAnyKeyword`. Она удивительным образом похожа на нашу фактическую функцию класса, но у нее нет тела функции. Определение функции `usingOptionalParameters` показывает, как использовать необязательный параметр в интерфейсе.

Однако определение интерфейса для функции `usingDefaultParameters` немного отличается от определения класса. Помните, что интерфейс определяет форму нашего класса или объекта и поэтому не может содержать переменные либо значения. Поэтому мы определили параметр `defaultArg1` как необязательный и оставили присвоение значения по умолчанию самой реализации класса. Определение функции `usingRestSyntax` содержит синтаксис оставшихся параметров, а определение функции `usingFunctionCallbacks` показывает, как определить сигнатуру функции обратного вызова. Они в значительной степени идентичны сигнатурам функций класса.

Единственное, чего не хватает в этом интерфейсе, – сигнатуры функции-конструктора. Интерфейсы не могут включать в себя эти сигнатуры.

Давайте посмотрим, почему это вызывает ошибки на нашем шаге компиляции. Предположим, что мы должны были включить определение функции-конструктора в интерфейс `IComplexType`:

```
interface IComplexType {
   constructor(arg1: any, arg2: any);
}
```

Тогда компилятор TypeScript выдаст ошибку:

```
error TS2420: Class 'ComplexType' incorrectly implements
interface 'IComplexType'.
Types of property 'constructor' are incompatible.
```

Эта ошибка показывает, что когда мы используем функцию-конструктор, возвращаемый тип конструктора неявно типизируется компилятором TypeScript. Следовательно, возвращаемым типом конструктора `IComplexType` будет `IComplexType`, а возвращаемым типом конструктора `ComplexType` будет `ComplexType`. Даже если функция `ComplexType` реализует интерфейс `IComplexType`, на самом деле это два разных типа. Поэтому сигнатуры конструктора всегда будут несовместимы, и они не допустимы в определениях интерфейса.

Модификаторы класса

Как мы кратко обсуждали во вводной главе, TypeScript использует модификаторы доступа `public` и `private`, чтобы пометить переменные и функции класса как открытые (`public`) или закрытые (`private`). Кроме того, мы также можем использовать модификатор доступа protected, о котором мы поговорим чуть позже.

Доступ к свойству открытого класса может быть получен из любого вызывающего кода. Рассмотрим следующий код:

```
class ClassWithPublicProperty {
   public id: number | undefined; }
```

```
let publicAccess = new ClassWithPublicProperty();
publicAccess.id = 10;
```

Здесь мы определили класс с именем `ClassWithPublicProperty`, у которого есть единственное свойство `id`. Далее мы создаем экземпляр этого класса с именем `publicAccess` и присваиваем значение `10` свойству `id` этого экземпляра. Давайте теперь рассмотрим, как маркировка свойства с помощью слова `private` повлияет на доступ к этому свойству:

```
class ClassWithPrivateProperty {
   private id: number;
   constructor(_id : number) {
      this.id = _id;
   }
}
let privateAccess = new ClassWithPrivateProperty(10);
privateAccess.id = 20;
```

Здесь мы определили класс с именем `ClassWithPrivateProperty`, у которого есть единственное свойство `id` и который теперь помечен как `private`. У этого класса также есть функция-конструктор, которая принимает один аргумент с именем `_id` и присваивает значение этого аргумента свойству `id`. Обратите внимание, что в ходе присваивания мы используем синтаксис `this.id`.

Затем мы создаем экземпляр этого класса с именем `privateAccess`, а после пытаемся присвоить значение `20` закрытому свойству `id`. Этот код, однако, выдаст ошибку:

```
error TS2341: Property 'id' is private and only accessible
within class 'ClassWithPrivateProperty'.
```

Как видно из сообщения, TypeScript не разрешит присваивание свойству `id` этого класса за пределами самого класса, поскольку мы пометили его как `private`. Обратите внимание, что мы можем присвоить значение свойству `id` из класса, как мы это делали в функции-конструкторе.

Функции класса являются открытыми по умолчанию. Если не указать модификатор доступа `private` для свойств или функций, их уровень доступа по умолчанию станет открытым. Классы также могут помечать функции и свойства как защищенные (`protected`), но мы рассмотрим это ключевое слово чуть позже, когда будем обсуждать наследование.

Модификаторы доступа конструктора

TypeScript также использует сокращенную версию функции-конструктора, позволяя указывать параметры с модификаторами доступа непосредственно в конструкторе. Рассмотрим следующий код:

```
class classWithAutomaticProperties {
   constructor(public id: number, private name: string){
   }
}

let myAutoClass = new classWithAutomaticProperties(1, "className");

console.log(`myAutoClass id: ${myAutoClass.id}`);
console.log(`myAutoClass.name: ${myAutoClass.name}`);
```

В этом фрагменте кода мы определяем класс с именем `ClassWithAutomaticProperties`. Функция-конструктор использует два аргумента – `id` типа `number` и `name` типа `string`. Обратите внимание, однако, что модификаторы доступа `public` для `id` и `private` для name определяются непосредственно в функции-конструкторе. Этот сокращенный вариант автоматически создает открытое свойство `id` в классе `ClassWithAutomaticProperties`, а также закрытое свойство `name`.

Этот сокращенный синтаксис доступен только в функции-конструкторе.

Затем мы создаем переменную `myAutoClass` и присваиваем ей новый экземпляр класса `ClassWithAutomaticProperties`. После создания экземпляра этого класса он автоматически получает два свойства: свойство `id` типа `number`, которое является открытым, и свойство `name` типа `string`, которое является закрытым. Однако при компиляции предыдущего кода появится ошибка:

```
Property 'name' is private and only accessible within class
'classWithAutomaticProperties'.
```

Эта ошибка говорит о том, что автоматическое свойство name объявлено как private и поэтому недоступно для кодирования вне самого класса.

Хотя эта техника сокращенной записи при создании автоматических переменных-членов существует, она может затруднить чтение кода. По мнению автора, как правило, лучше использовать более подробные определения классов, которые не применяют данную технику. Если вы не будете прибегать к ней, а вместо этого перечислите все свойства в верхней части класса, человек, который читает код, сразу увидит, какие переменные использует этот класс, а также являются ли они открытыми или закрытыми. Использование синтаксиса автоматических свойств конструктора несколько скрывает эти параметры, заставляя разработчиков иногда перечитывать код, чтобы понять его. Однако какой бы синтаксис вы ни выбрали, постарайтесь привести его к стандарту кодирования и использовать один и тот же синтаксис на протяжении всей своей кодовой базы.

Свойство readonly

В дополнение к модификаторам доступа `public`, `private` и `protected` мы также можем пометить свойство класса как `readonly`. Это означает, что после того, как значение свойства установлено, оно не может быть изменено ни самим классом, ни пользователями этого класса. Есть только одно место, где может быть установлено свойство `readonly`, – внутри самой функции-конструктора. Рассмотрим следующий код:

```
class ClassWithReadOnly {
   readonly name: string;

   constructor(_name : string) {
      this.name = _name;
   } setReadOnly(_name: string) {
      // генерирует ошибку компиляции
      this.name = _name;
   }
}
```

Здесь мы определили класс с именем `ClassWithReadOnly`, у которого есть свойство name типа `string`, помеченное ключевым словом `readonly`. Функция-конструктор устанавливает это значение. Затем мы определили вторую функцию с именем `setReadOnly`, где пытаемся установить свойство `readonly`. Этот код выдаст ошибку:

```
error TS2540: Cannot assign to 'name' because it is a constant
or a readonly property.
```

Это сообщение говорит нам, что единственное место, где можно установить свойство `readonly`, находится в функции-конструкторе.

Методы доступа к свойствам класса

ECMAScript 5 представляет концепцию методов доступа к свойствам. Метод доступа – это просто функция, которая вызывается, когда пользователь нашего класса либо устанавливает свойство, либо получает его. Это означает, что мы можем определить, когда пользователи нашего класса либо получают (`get`), либо устанавливают (`set`) свойство, и может использоваться в качестве пускового механизма для другой логики. Чтобы использовать методы доступа, мы создаем пару функций `get` и `set` (с одинаковым именем), чтобы получить доступ к внутреннему свойству. Эту концепцию лучше всего понять с помощью примера:

```
class ClassWithAccessors {
   private _id : number | undefined;
   get id() {
```

```
        console.log(`inside get id()`);
        return <number>this._id;
    }


    set id(value:number) {
        console.log(`inside set id()`);
        this._id = value;
    }
}
```

У этого класса есть закрытое свойство `_id` и две функции, обе из которых называются `id`. Первая из этих функций начинается с ключевого слова `get`. Эта функция вызывается, когда пользователь нашего класса получает или читает свойство. В нашем примере функция `get` записывает сообщение об отладке в консоль, а затем просто возвращает значение внутреннего свойства `id`.

У второй функции есть префикс с ключевым словом `set`, и она принимает параметр `value`. Эта функция вызывается, когда пользователь нашего класса присваивает значение или устанавливает свойство. В нашем примере мы просто записываем сообщение в консоль, а затем устанавливаем внутреннее свойство `_id`. Обратите внимание, что оно является закрытым и поэтому не доступно за пределами самого класса.

Теперь мы можем использовать этот класс следующим образом:

```
var classWithAccessors = new ClassWithAccessors();
classWithAccessors.id = 2;
console.log(`id property is set to ${classWithAccessors.id}`);
```

Здесь мы создали экземпляр класса и назвали его `classWithAccessors`. Обратите внимание, что мы не используем две отдельные функции `get` и `set`. Мы просто используем их как единичное свойство `id`. Когда мы присвоим этому свойству значение, среда выполнения ECMAScript 5 вызовет функцию `set id (value)`, а когда мы получим это свойство, среда выполнения вызовет функцию `get id()`. Вывод этого кода выглядит так:

```
inside set id()
inside get id()
id property is set to 2
```

Использование функций getter и setter позволяет нам подключаться к свойствам класса и выполнять код при обращении к этим свойствам класса.

Это свойство доступно только при использовании ECMAScript 5 и выше. Имейте в виду, что некоторые браузеры не поддерживают ECMAScript 5 (например, Internet Explorer 8) и будут выдавать ошибки при попытке использовать методы доступа к классам.

Статические функции

Статические функции – это функции, которые можно вызывать в классе без необходимости вначале создавать экземпляр класса. Эти функции почти глобальны по своей природе и должны вызываться путем добавления префикса имени функции к имени класса. Рассмотрим следующее определение класса:

```
class StaticClass {
   static printTwo() {
      console.log(`2`);
   }
}
StaticClass.printTwo();
```

Это определение включает в себя одну функцию с именем `printTwo`, которая помечена как статическая. Как видно из последней строки кода, мы можем вызывать эту функцию, не создавая новый экземпляр класса `StaticClass`. Мы можем просто вызывать функцию напрямую, если добавляем префикс к имени класса.

Статические свойства

Подобно статическим функциям, у классов также могут быть статические свойства. Если свойство класса помечено как `static`, то у каждого экземпляра этого класса будет одинаковое значение свойства. Другими словами, у всех экземпляров одного и того же класса будет общее статическое свойство. Рассмотрим следующий код:

```
class StaticProperty {
   static count = 0;
   updateCount() {
      StaticProperty.count++;
   }
}
```

Данное определение класса использует ключевое слово `static` для свойства `count`. У него есть единственная функция `updateCount`, которая увеличивает статическое свойство `count`. Обратите внимание на синтаксис этой функции. Обычно мы используем ключевое слово `this` для доступа к свойствам класса.

Здесь, однако, нам нужно сослаться на полное имя свойства, включая имя класса, то есть `StaticProperty.count`, чтобы получить доступ к этому свойству. Это похоже на то, что мы видели в случае со статическими функциями. Затем мы можем использовать этот класс следующим образом:

```
let firstInstance = new StaticProperty();
```

```
console.log(`StaticProperty.count = ${StaticProperty.count}`);
firstInstance.updateCount();

console.log(`StaticProperty.count = ${StaticProperty.count}`);
let secondInstance = new StaticProperty();

secondInstance.updateCount();
console.log(`StaticProperty.count = ${StaticProperty.count}`);
```

Этот фрагмент кода начинается с нового экземпляра класса `StaticProperty` с именем `firstInstance`. Затем мы записываем значение `StaticProperty.count` в консоль, а после вызываем функцию `updateCount` для этого экземпляра класса и снова записываем значение `StaticProperty.count` в консоль. Затем мы создаем еще один экземпляр этого класса с именем `secondInstance`, вызываем функцию `updateCount` и записываем значение `StaticProperty.count` в консоль. Этот фрагмент кода выведет следующее:

```
StaticProperty.count = 0
StaticProperty.count = 1
StaticProperty.count = 2
```

Видно, что статическое свойство с именем `StaticCount.count` действительно совместно используется двумя экземплярами одного и того же класса, а именно `firstInstance` и `secondInstance`. Оно начинается с `0` и увеличивается при вызове `firstInstance.updateCount`. Когда мы создаем второй экземпляр класса, он также сохраняет свое первоначальное значение `1` для этого экземпляра. Когда `secondInstance` обновит это количество, оно также будет обновлено для `firstInstance`.

Пространства имен

При работе с крупными проектами, особенно внешними библиотеками, может наступить момент, когда два класса или интерфейса совместно используют одно и то же имя. Это, очевидно, приведет к ошибкам компиляции. TypeScript использует концепцию пространств имен для учета подобных ситуаций.

Давайте посмотрим на синтаксис, используемый для пространств имен:

```
namespace FirstNameSpace {
   class NotExported { }
   export class NameSpaceClass {
      id: number | undefined;
   }
}
```

Здесь мы определяем пространство имен с помощью ключевого слова `namespace`. Мы назвали его `FirstNameSpace`. Объявление пространства имен аналогично объ-

явлению класса в том смысле, что оно ограничено открывающей и закрывающей фигурной скобкой, то есть { запускает пространство имен и } закрывает его. В этом пространстве имен определены два класса: `NotExported` и `NameSpaceClass`.

При использовании пространств имен определение класса не будет видно за пределами пространства имен, если только мы специально не разрешим этого с помощью ключевого слова export. Чтобы создать классы, которые определены в пространстве имен, мы должны обратиться к классу, используя полное имя пространства имен. Давайте посмотрим, как мы будем создавать экземпляры этих классов:

```
let firstNameSpace = new FirstNameSpace.NameSpaceClass();
let notExported = new FirstNameSpace.NotExported();
```

Здесь мы создаем экземпляр `NameSpaceClass` и экземпляр класса `NotExported`. Обратите внимание, что нам нужно использовать полное имя пространства имен для правильного обращения к этим классам, то есть new `FirstNameSpace.NameSpaceClass()`. Поскольку класс `NotExported` не использовал ключевое слово `export`, последняя строка этого кода выдаст ошибку:

```
error TS2339: Property 'NotExported' does not exist on type
'typeof FirstNameSpace'.
```

Теперь мы можем ввести второе пространство имен:

```
namespace SecondNameSpace {
   export class NameSpaceClass {
      name: string | undefined;
   }
}

let secondNameSpace = new SecondNameSpace.NameSpaceClass();
```

Это пространство имен также экспортирует класс `NameSpaceClass`. В последней строке данного фрагмента кода мы снова создаем экземпляр этого класса, используя полное имя пространства имен, а именно new `FirstNameSpace.NameSpaceClass()`. Использование того же имени класса в этом экземпляре не приведет к ошибкам компиляции, поскольку каждый класс (с префиксом имени пространства имен) рассматривается компилятором как отдельное имя класса.

Наследование

Наследование – еще одна парадигма, которая является одним из краеугольных камней объектно-ориентированного программирования. Наследование означает, что объект использует другой объект в качестве своего базового типа, тем самым наследуя все характеристики базового объекта, включая все свойства и функции. И интерфейсы, и классы могут использовать наследование. Интер-

фейс или класс, унаследованный от него, известен как базовый интерфейс или базовый класс, а интерфейс или класс, который выполняет наследование, известен как производный интерфейс или производный класс. TypeScript реализует наследование, используя ключевое слово `extends`.

Наследование интерфейса

В качестве примера наследования интерфейса рассмотрим приведенный ниже код:

```
interface IBase {
   id: number | undefined;
}

interface IDerivedFromBase extends IBase {
   name: string | undefined;
}

class InterfaceInheritanceClass implements IDerivedFromBase {
   id: number | undefined;
   name: string | undefined;
}
```

Вначале у нас идет интерфейс под названием `IBase`, который определяет свойство `id` типа `number` или `undefined`. Наше второе определение интерфейса, `IDerivedFromBase`, наследует (`extends`) от `IBase`, а следовательно, автоматически включает в себя свойство `id`. Затем интерфейс `IDerivedFromBase` определяет свойство `name` типа `string` или `undefined`.

Поскольку интерфейс `IDerivedFromBase` наследуется от `IBase`, у него фактически два свойства – `id` и `name`. Далее у нас идет определение класса `InterfaceInheritanceClass`, которое реализует интерфейс `IDerivedFromBase`. Следовательно, этот класс должен определять как `id`, так и свойство `name`, чтобы успешно реализовать все свойства интерфейса `IDerivedFromBase`. Хотя у нас есть только показанные свойства в этом примере, те же правила применяются для функций.

Наследование классов

Классы также могут использовать наследование, как и интерфейсы. Используя наши определения интерфейсов `IBase` и `IDerivedFromBase`, приведенный ниже код показывает пример наследования классов:

```
class BaseClass implements IBase {
   id: number | undefined;
}
```

```
class DerivedFromBaseClass
extends BaseClass
implements IDerivedFromBase {
   name: string | undefined;
}
```

Первый класс, `BaseClass`, реализует интерфейс `IBase` и как таковой требуется только для определения свойства `id`, типа `number` или `undefined`. Второй класс, `DerivedFromBaseClass`, наследует от класса `BaseClass` (с использованием ключевого слова `extends`), но также реализует интерфейс `IDerivedFromBase`. Поскольку `BaseClass` уже определяет свойство `id`, требуемое в интерфейсе `IDerivedFromBase`, единственное другое свойство, которое должен реализовать класс `DerivedFromBaseClass`, – это свойство `name`. Следовательно, нам нужно включить определение только этого свойства в класс `DerivedFromBaseClass`.

TypeScript не поддерживает концепцию множественного наследования. Множественное наследование означает, что один класс может быть получен из множества базовых классов. TypeScript поддерживает только одиночное наследование, и поэтому у любого класса может быть только один базовый класс.

Однако класс может реализовывать множество интерфейсов:

```
interface IFirstInterface {
   id : number | undefined;
}
interface ISecondInterface {
   name: string | undefined;
}

class MultipleInterfaces
implements IFirstInterface, ISecondInterface {
   id: number | undefined;
   name: string | undefined;
}
```

Здесь мы определили два интерфейса с именами `IFirstInterface` и `ISecondInterface`.

Далее следует класс с именем `MultipleInterfaces`, который реализует оба интерфейса. Это означает, что класс `MultipleInterfaces` должен реализовывать свойство `id`, чтобы удовлетворять интерфейсу `IFirstInterface`, и свойство name для удовлетворения интерфейсу `IFirstInterface`.

Ключевое слово super

При использовании наследования и базовый, и производный классы могут иметь одно и то же имя функции. Это чаще всего наблюдается при работе с конструкто-

рами классов. Если у базового класса есть определенный конструктор, то производному классу может потребоваться обратиться к конструктору базового класса и пройти аргументы. Эта техника носит название переопределение конструктора. Другими словами, конструктор производного класса переопределяет или заменяет конструктор базового класса. TypeScript использует ключевое слово super, позволяющее вызывать функцию базового класса с тем же именем. Рассмотрим следующие классы:

```
class BaseClassWithConstructor {
   private id: number;
   constructor(_id: number) {
      this.id = _id;
   }
}

class DerivedClassWithConstructor
extends BaseClassWithConstructor {
   private name: string;
   constructor(_id: number, _name: string) {
      super(_id);
      this.name = _name;
   }
}
```

Здесь мы определяем класс с именем `BaseClassWithConstructor`, который содержит закрытое свойство `id`. У этого класса есть функция-конструктор, которая требует аргумента `_id`. Второй класс, `DerivedClassWithConstructor`, наследует от класса `BaseClassWithConstructor`. Конструктор `DerivedClass WithConstructor` принимает аргументы `_id` и `_name`. Тем не менее ему нужно передать входящий аргумент `_id` через базовый класс. Вот тут вступает в действие вызов super. Ключевое слово super вызывает функцию в базовом классе, имя которой совпадает с именем функции в производном классе. Первая строка функции-конструктора `DerivedClassWithConstructor` показывает вызов, используя ключевое слово `super`, передавая полученный аргумент `id` конструктору базового класса.

Переопределение функции

Однако конструктор класса – это просто функция. Так же, как мы можем использовать ключевое слово super в конструкторе, мы можем использовать ключевое слово super, когда базовый класс и его производный класс используют то же имя функции. Эта техника носит название переопределение функций.

Другими словами, у производного класса есть имя функции, совпадающее с именем функции базового класса, и он переопределяет это определение функции. Рассмотрим приведенный ниже код:

```
class BaseClassWithFunction {
   public id : number | undefined;
   getProperties() : string {
      return `id: ${this.id}`;
   }
}

class DerivedClassWithFunction
extends BaseClassWithFunction {
   public name: string | undefined;
   getProperties() : string {
      return `${super.getProperties()}`+ ` , name: ${this.name}`;
   }
}
```

В данном примере мы определили класс с именем `BaseClassWithFunction`, с открытым свойством `id` и функцией `getProperties`, которая просто возвращает строковое представление свойств класса. Однако класс `DerivedClassWithFunction` также включает в себя функцию `getProperties`. Эта функция является переопределением функции базового класса `getProperties`. Чтобы вызвать функцию базового класса, нам нужно использовать ключевое слово super, как показано в вызове `super.getProperties`.

Давайте посмотрим, как мы будем использовать эти классы:

```
var derivedClassWithFunction = new DerivedClassWithFunction();
derivedClassWithFunction.id = 1;
derivedClassWithFunction.name = "derivedName";
console.log(derivedClassWithFunction.getProperties());
```

Этот код создает переменную с именем `derivedClassWithFunction`, которая является экземпляром класса `DerivedClassWithFunction`. Затем он устанавливает свойства `id` и `name` и записывает результат вызова функции `getProperties` в консоль. В результате мы получаем следующее:

```
id: 1 , name: derivedName
```

Результаты показывают, что функция `getProperties` переменной `derivedClass WithFunction` будет вызывать функцию базового класса `getProperties`, как и ожидалось.

Члены класса protected

При использовании наследования иногда логично пометить определенные функции и свойства как доступные только внутри самого класса или доступные любому классу, производному от него. Однако использование ключевого слова `private` не будет работать в этом случае, так как член закрытого класса скрыт

даже от производных классов. Для таких случаев TypeScript использует ключевое слово `protected`. Рассмотрим два класса:

```
class ClassUsingProtected {
   protected id : number | undefined;
   public getId() {
      return this.id;
   }
}

class DerivedFromProtected
extends ClassUsingProtected {
   constructor() {
      super();
      this.id = 0;
   }
}
```

Мы начнем с класса `ClassUsingProtected`, у которого есть свойство `id`, помеченное как `protected`, и открытая функция `getId`. Следующий класс `DerivedFromProtected` наследует от `ClassUsingProtected`, и у него есть одна функция-конструктор. Обратите внимание, что в этой функции мы вызываем `this.id = 0`, чтобы установить для защищенного свойства `id` значение `0`. Опять же, производный класс имеет доступ к защищенным переменным-членам. Теперь давайте попробуем получить доступ к свойству `id` вне класса:

```
var derivedFromProtected = new DerivedFromProtected();
derivedFromProtected.id = 1;
console.log(`getId returns: ${derivedFromProtected.getId()}`);
```

Здесь мы создаем экземпляр класса `DerivedFromProtected` и пытаемся присвоить значение его защищенному свойству `id`. Компилятор выдаст сообщение об ошибке:

```
error TS2445: Property 'id' is protected and only accessible
within class 'ClassUsingProtected' and its subclasses.
```

Таким образом, свойство `id` действует как закрытое свойство за пределами класса `ClassUsingProtected`, но по-прежнему разрешает доступ к нему внутри класса и любому классу, производному от него.

Абстрактные классы

Еще одним фундаментальным принципом объектно-ориентированного проектирования является концепция абстрактных классов. Абстрактный класс – это класс, который не предполагает создания экземпляров. Другими словами, это класс, который предназначен для наследования. Абстрактные классы, иногда называемые

абстрактными базовыми классами, часто используются для предоставления набора базовых функций или свойств, которые распределяются между группой похожих классов. Они похожи на интерфейсы в том плане, что не предполагают создания экземпляров, но у них могут быть реализации функций, которых нет в интерфейсах.

Абстрактные классы – это метод, позволяющий повторно использовать код в группах похожих объектов. Рассмотрим два класса:

```
class Employee {
   public id: number | undefined;
   public name: string | undefined;
   printDetails() {
      console.log(`id: ${this.id}`+ `, name ${this.name}`);
   }
}

class Manager {
   public id: number | undefined;
   public name: string | undefined;
   public Employees: Employee[] | undefined;
   printDetails() {
      console.log(`id: ${this.id} ` + `, name ${this.name},
      ` + ` employeeCount ${this.Employees.length}`);
   }
}
```

Вначале идет класс `Employee`, у которого есть свойства `id` и `name`, а также функция `printDetails`. Следующий класс называется `Manager`, и он очень похож на класс `Employee`. У него также есть свойства `id` и `name`, а еще дополнительное свойство `Employees` – список сотрудников, за которыми наблюдает этот менеджер. У этих двух классов много общего кода. У обоих есть свойства `id` и `name`, и у обоих есть функция `printDetails`. Использование абстрактного базового класса для этих классов преодолевает данную проблему с помощью общих свойств и кода. Давайте перепишем их, используя концепцию абстрактного базового класса:

```
abstract class AbstractEmployee {
   public id: number | undefined;
   public name: string | undefined;
   abstract getDetails(): string;
   public printDetails() {
      console.log(this.getDetails());
   }
}

class NewEmployee extends AbstractEmployee {
   getDetails(): string {
      return `id : ${this.id}, name : ${this.name}`;
```

```
    }
}

class NewManager extends NewEmployee {
    public Employees: NewEmployee[] | undefined;
    getDetails() : string {
        return super.getDetails()
        + `, employeeCount ${this.Employees.length}`;
    }
}
```

Здесь мы определили абстрактный класс с именем `AbstractEmployee`, который включает в себя свойства `id` и `name`, общие для менеджеров и сотрудников. Затем мы определяем так называемую абстрактную функцию с именем `getDetails`. Использование абстрактной функции означает, что любой класс, производный от этого абстрактного класса, должен реализовывать эту функцию. Далее мы определяем функцию `printDetails` для записи сведений о классе `AbstractEmployee` в консоль. Обратите внимание, что мы вызываем абстрактную функцию `getDetails` из абстрактного класса. Это означает, что код в функции `printDetails` будет вызывать фактическую реализацию функции в производном классе.

Второй класс, `NewEmployee`, наследует от класса `AbstractEmployee`. Таким образом, он должен реализовывать функцию `getDetails`, которая была помечена как абстрактная в базовом классе. Функция `getDetails` возвращает строковое представление свойств `id` и `name` класса `NewEmployee`.

Далее у нас идет класс с именем `NewManager`, производный от `NewEmployee`. Следовательно, у класса `NewManager` также есть свойства `id` и `name`, а кроме этого, дополнительное свойство с именем `Employees`. Поскольку этот класс уже является производным от `NewEmployee`, ему не обязательно снова определять функцию `getDetails`. Он может просто использовать версию функции `getDetails`, которую предоставляет класс `NewEmployee`. Обратите внимание, однако, что мы фактически определили функцию `getDetails` в классе `NewManager`. Эта функция вызывает функцию базового класса `getDetails` через ключевое слово `super`, а затем добавляет дополнительную информацию о свойстве `Employees`. Давайте посмотрим, что происходит, когда мы создаем и используем эти классы:

```
var employee = new NewEmployee();
employee.id = 1;
employee.name = "Employee Name";
employee.printDetails();
```

Здесь мы создали экземпляр класса `NewEmployee` с именем `employee`, установили его свойства `id` и `name` и вызвали функцию `printDetails` из абстрактного класса. Напомним, что абстрактный класс затем вызовет реализацию функции `getDetails`, которую мы предоставили в классе `NewEmployee`, поэтому вывод консоли будет таким:

```
id: 1, name : Employee Name
```

Теперь давайте используем наш класс `NewManager` аналогичным образом:

```
var manager = new NewManager();
manager.id = 2;
manager.name = "Manager Name";
manager.Employees = [];
manager.printDetails();
```

Здесь мы создали экземпляр класса `NewManager` с именем `manager` и установили его свойства `id` и `name`, как и раньше. Поскольку в этом классе также есть массив `Employees`, мы устанавливаем свойство `Employees` в пустой массив. Однако обратите внимание, что происходит, когда мы вызываем функцию `printDetails` в последней строке. Вывод будет следующим:

```
id: 2, name : Manager Name, employeeCount 0
```

Реализация абстрактного класса функции `printDetails` вызывает функцию `getDetails` производного класса. Поскольку класс `NewManager` также определяет функцию `getDetails`, абстрактный класс будет вызывать эту функцию для экземпляра `NewManager`.

Затем функция `getDetails` вызывает реализацию базового класса, то есть экземпляр `NewEmployee` функции `getDetails()`, а после добавляет информацию о количестве сотрудников.

Использование абстрактных классов и наследования позволяет писать код более понятным и многократно используемым способом. Абстракция, наследование, полиморфизм и инкапсуляция являются основой правильных принципов объектно-ориентированного проектирования. Как мы уже видели, язык TypeScript дает нам возможность объединить каждый из этих принципов, чтобы помочь написать качественный, чистый код JavaScript.

Замыкания JavaScript

Прежде чем продолжить, давайте кратко рассмотрим, как TypeScript реализует классы в сгенерированном JavaScript ES3 или ES5 с помощью метода под названием замыкания. Как мы упоминали в `главе 1` *«Инструменты TypeScript и параметры фреймворков»*, замыкание – это функция, которая ссылается на независимые переменные. Эти переменные, по существу, запоминают среду, в которой они были созданы. Рассмотрим приведенный ниже код:

```
function TestClosure(value) {
  this._value = value;
  function printValue() {
    console.log(this._value);
```

```
    }
    return printValue;
}

var myClosure = TestClosure(12);
myClosure;
```

Здесь у нас есть функция `TestClosure`, которая принимает один параметр с именем `value`. Тело функции сначала присваивает аргумент `value` внутреннему свойству `this._value`, а затем определяет внутреннюю функцию с именем `printValue`. Функция `printValue` просто записывает значение `this._value` в консоль. Интересный момент – последняя строка в функции `TestClosure` – мы возвращаем функцию `printValue`.

Теперь взгляните на последние две строки фрагмента кода. Мы создаем переменную с именем `myClosure` и присваиваем ей результат вызова функции `TestClosure`. Обратите внимание, что поскольку мы возвращаем функцию `printValue` из функции `TestClosure`, это также делает переменную `myClosure` функцией. Когда мы выполним эту функцию в последней строке фрагмента, она выполнит внутреннюю функцию `printValue`, но запомнит начальное значение `12`, которое использовалось при создании переменной `myClosure`. Вывод последней строки кода запишет значение `12` в консоль.

Это основная природа замыканий. Закрытие – это особый вид объекта JavaScript, который объединяет функцию с исходной средой, в которой она была создана. В нашем предыдущем примере, поскольку мы сохранили все, что было передано через аргумент `value`, в локальную переменную с именем `this._value`, JavaScript запоминает среду, в которой было создано замыкание. Другими словами, все, что было назначено свойству `this._value` на момент создания, будет запомнено и может быть использовано позже.

Имея это в виду, давайте посмотрим на код JavaScript, который генерируется компилятором TypeScript для класса `BaseClassWithConstructor`, с которым мы только что работали:

```
var BaseClassWithConstructor = (function () {
    function BaseClassWithConstructor(_id) {
        this.id = _id;
    }
    return BaseClassWithConstructor;
})();
```

Наше замыкание начинается с `function()  {` в первой строке и заканчивается фигурной скобкой `}` в последней строке.

Замыкание сначала определяет функцию, которая будет использоваться в качестве конструктора: `BaseClassWithConstructor(_id)`.

Оно окружено открывающей скобкой (в первой строке и закрывающей скобкой) в последней строке, определяя выражение функции JavaScript. Это выражение функции затем немедленно выполняется двумя последними скобками, `();`. Подобная техника немедленного выполнения функции известна как выражение немедленно вызываемой функции. Наше предыдущее выражение затем назначается переменной с именем `BaseClassWithConstructor`, что делает ее объектом первого класса JavaScript, который можно создать с помощью ключевого слова `new`. Таким образом, TypeScript реализует классы в JavaScript.

Реализация базового кода JavaScript, который TypeScript использует для определений классов, на самом деле является хорошо известным шаблоном JavaScript под названием шаблон модуля. Хорошая новость: глубокое знание замыканий, как их писать и как использовать шаблон модуля для определения классов – обо всем этом позаботится компилятор TypeScript, что позволит нам сосредоточиться на принципах ООП без необходимости писать замыкания JavaScript с использованием шаблонного кода данного типа.

instanceof

TypeScript также позволяет проверять, был ли объект создан из определенного класса, используя ключевое слово instanceof. Чтобы проиллюстрировать это, рассмотрим следующий набор классов:

```
class A { }
class BfromA extends A { }
class CfromA extends A { }class DfromC extends CfromA { }
```

Здесь мы определили четыре класса, начиная с простого класса с именем `A`. Затем мы определяем два класса, производных от `A`, `BfromA` и `CfromA`. Последним определен класс с именем `DfromC`, который является производным от класса `CfromA`. Давайте теперь воспользуемся ключевым словом `instanceof`, чтобы узнать, что думает компилятор об этой структуре наследования:

```
function checkInstanceOf(value: A | BfromA | CfromA | DfromC) {
   console.log(`checking instanceof :`)
   if (value instanceof A) {
      console.log(`found instanceof A`);
   }
   if (value instanceof BfromA) {
      console.log(`found instanceof BfromA`);
   }
   if (value instanceof CfromA) {
      console.log(`found instanceof CFromA`);
   }
   if (value instanceof DfromC) {
      console.log(`found instanceof DfromC`);
```

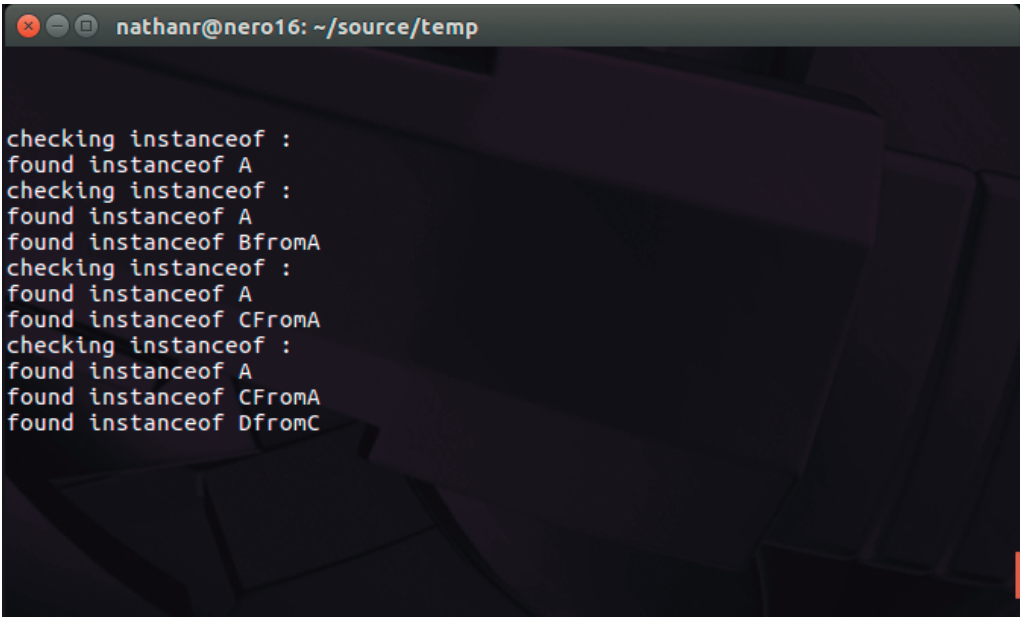
```
    }
}
```

Здесь мы создали функцию с именем `checkInstanceOf`, у которой есть единственный параметр, который может быть любым из наших четырех типов классов. Как только мы попадаем внутрь функции, мы записываем сообщение в консоль, указывая, что приступаем к проверке входного аргумента. Далее идут четыре оператора `if`, каждый из которых проверяет, соответствует ли тип аргумента значения любому из четырех типов классов.

Давайте используем эту функцию:

```
checkInstanceOf(new A());
checkInstanceOf(new BfromA());
checkInstanceOf(new CfromA());
checkInstanceOf(new DfromC());
```

Здесь мы вызываем функцию `checkInstanceOf` с экземпляром каждого из четырех классов. Вывод этого кода выглядит так:



Видно, что если мы вызываем эту функцию с классом типа A, наш код правильно распознает это, поскольку `if (value instanceof A)` возвращает значение `true`. Интересно, что когда мы отправляем экземпляр класса `BfromA`, то видим, что операторы `instanceof A` и `instanceof BfromA` возвращают значение `true`. Это означает, что компилятор правильно определяет, что класс `BfromA` был получен из A. Та же логика применима к классу `CfromA`. Посмотрите, как класс `DfromC`

заставит три оператора `instanceof` вернуть значение `true`. Наш код обнаруживает, что `DfromC` является производным от `CfromA` и что, поскольку `CfromA` происходит от `A`, `DfromC` также является экземпляром типа `A`.

Использование интерфейсов, классов и наследования – шаблон проектирования Factory

Чтобы проиллюстрировать, как можно использовать интерфейсы и классы в крупном проекте TypeScript, мы кратко рассмотрим один очень известный объектно-ориентированный шаблон проектирования – Factory.

Бизнес-требования

В качестве примера предположим, что наш бизнес-аналитик предъявляет нам следующие требования.

1. От вас требуется классифицировать людей по дате их рождения по трем возрастным группам: Младенцы, Дети и Взрослые.
2. Укажите с помощью значений `true` или `false`, достигли ли они совершеннолетия, чтобы подписать контракт.
3. Человек считается младенцем, если он моложе двух лет.
4. Младенцы не могут подписывать контракты.
5. Человек считается ребенком, если он моложе 18.
6. Дети также не могут подписывать контракты.
7. Человек считается совершеннолетним, если он старше 18.
8. Только взрослые могут подписывать контракты.
9. В целях отчетности каждый тип лица должен иметь возможность вывести свои данные.

Сюда должны быть включены:

- дата рождения;
- категория лица;
- могут ли они подписывать контракты или нет.

Что делает шаблон проектирования Factory

Шаблон проектирования Factory использует класс `Factory` для возврата экземпляра одного из нескольких возможных классов на основе предоставленной ему информации.

Суть этого шаблона состоит в том, чтобы поместить логику принятия решения относительно того, какой тип класса следует создать, в отдельный класс под названием класс `Factory`. Затем класс `Factory` вернет один из нескольких классов, которые являются тонкими вариациями друг друга, которые будут делать немного разные вещи в зависимости от их специальности. Поскольку мы не знаем, какой тип класса будет возвращать шаблон Factory, нам нужен способ работы с любой вариацией различных типов возвращаемых классов. Звучит как идеальный сценарий для интерфейса.

Для реализации необходимой бизнес-функциональности мы создадим классы `Infant`, `Child` и `Adult`. Классы `Infant` и `Child` вернут значение `false`, когда их спросят, могут ли они подписывать контракты, а класс `Adult` вернет значение `true`.

Интерфейс IPerson

Согласно нашим требованиям, экземпляр класса, возвращаемый Factory, должен уметь делать две вещи: выводить категорию лица в требуемом формате и сообщать нам, могут ли они подписывать контракты. Начнем с определения перечисления и интерфейса, чтобы удовлетворить это требование:

```
enum PersonCategory {
   Infant,
   Child,
   Adult,
   Undefined
}

interface IPerson {
   Category: PersonCategory;
   canSignContracts(): boolean;
   printDetails();
}
```

Первым идет перечисление, использующееся для хранения допустимых значений `PersonCategory`, а именно `Infant`, `Child`, `Adult` или `Undefined`. Значение `Undefined` используется, чтобы указать, что мы еще не классифицировали лицо. Затем мы определяем интерфейс с именем `IPerson`, который содержит все функциональные возможности, общие для каждого типа лиц. Сюда входят `Category`, функция `canSignContracts`, которая возвращает значения `true` или `false`, и функция для вывода их данных `printDetails`.

Класс Person

Чтобы упростить наш код, мы создадим абстрактный базовый класс для хранения всего кода, который является общим для младенцев, детей и взрослых. Опять же,

абстрактные классы не предполагают создания экземпляров и как таковые предназначены для наследования. Назовем этот класс `Person`:

```
abstract class Person implements IPerson {
   Category: PersonCategory;
   private DateOfBirth: Date;

   constructor(dateOfBirth: Date) {
      this.DateOfBirth = dateOfBirth;
      this.Category = PersonCategory.Undefined;
   }

   abstract canSignContracts(): boolean
   printDetails() : void {
      console.log(`Person : `);
      console.log(`Date of Birth : `
         + `${this.DateOfBirth.toDateString()}`);
      console.log(`Category : `
         + `${PersonCategory[this.Category]}`);
      console.log(`Can sign : `
         + `${this.canSignContracts()}`);
   }
}
```

Абстрактный класс `Person` реализует интерфейс `IPerson`, и поэтому ему понадобятся три вещи: свойство `Category`, функция `canSignContracts` и функция `printDetails`. Чтобы вывести дату рождения человека, нам также нужно свойство `DateOfBirth`, которое мы установим в нашей функции-конструкторе.

Есть несколько интересных моментов касательно класса `Person`, которые следует отметить. Во-первых, свойство `DateOfBirth` было объявлено закрытым (`private`). Это означает, что единственный класс, который имеет доступ к свойству `DateOfBirth`, – это сам класс `Person`. В наших требованиях не упоминается использование даты рождения вне функции вывода, следовательно, нет необходимости получать доступ или изменять дату рождения, как только она была установлена.

Во-вторых, функция `canSignContracts` была помечена как абстрактная. Это означает, что любой класс, производный от этого класса, вынужден реализовать функцию `canSignContracts`, которая является именно тем, что мы хотели.

В-третьих, функция `printDetails` полностью реализована в этом абстрактном классе. Это означает, что единая функция вывода автоматически доступна для любого класса, наследованного от `Person`.

Классы специалистов

Теперь что касается трех классов специалистов. Все они были наследованы из базового класса `Person`:

```
class Infant extends Person {
   constructor(dateOfBirth: Date) {
      super(dateOfBirth);
      this.Category = PersonCategory.Infant;
   }
   canSignContracts(): boolean { return false; }
}

class Child extends Person {
   constructor(dateOfBirth: Date) {
      super(dateOfBirth);
      this.Category = PersonCategory.Child;
   }
   canSignContracts(): boolean { return false; }
}

class Adult extends Person {
   constructor(dateOfBirth: Date) {
      super(dateOfBirth);
      this.Category = PersonCategory.Adult;
   }
   canSignContracts(): boolean { return true; }
}
```

Каждый из этих классов использует наследование для расширения класса `Person`. Поскольку свойство `DateOfBirth` объявлено закрытым и, следовательно, видимо только для самого класса `Person`, мы должны передать его классу `Person` в каждом из наших конструкторов. Каждый конструктор также устанавливает свойство `Category` на основе типа класса. Наконец, каждый класс реализует абстрактную функцию `canSignContracts`.

Одним из преимуществ использования наследования, таким образом, является то, что определения фактических классов становятся очень простыми. По сути, наши классы делают только две вещи: правильно устанавливают свойство `Category` и определяют, могут ли они подписывать контракты.

Класс Factory

Давайте теперь перейдем к самому классу `Factory`. У этого класса есть одна четко определенная обязанность: учитывая дату рождения, выяснить, меньше ли она, чем два года назад, 18 лет назад, или больше, чем 18 лет назад. На основании этих решений возвращаем экземпляр класса `Infant`, `Child` или `Adult`:

```
class PersonFactory {
getPerson(dateOfBirth: Date) : IPerson {
   let dateNow = new Date(); // defaults to now.
```

```
    let currentMonth = dateNow.getMonth() + 1;
    let currentDate = dateNow.getDate();
    let dateTwoYearsAgo = new Date(
      dateNow.getFullYear() - 2, currentMonth, currentDate);
      let date18YearsAgo = new Date( dateNow.getFullYear() - 18,
        currentMonth, currentDate);
      if (dateOfBirth >= dateTwoYearsAgo) {
        return new Infant(dateOfBirth);
      }
      if (dateOfBirth >= date18YearsAgo) {
        return new Child(dateOfBirth);
      }
      return new Adult(dateOfBirth);
    }
  }
```

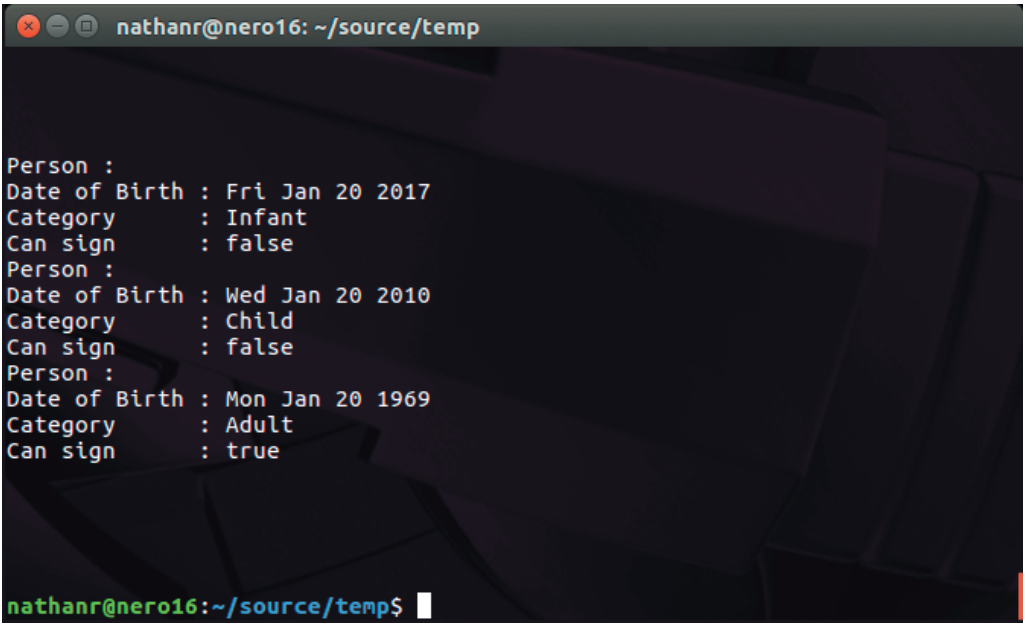
У класса `PersonFactory` есть только одна функция `getPerson`, которая возвращает объект типа `IPerson`. Функция создает переменную с именем `dateNow`, которая устанавливается на текущую дату. Затем мы находим текущий месяц и дату из переменной `dateNow`. Обратите внимание, что функция JavaScript, `getMonth`, возвращает `0–11`, а не `1–12`, поэтому мы исправляем это, добавляя `1`. Затем эта переменная `dateNow` используется для вычисления еще двух переменных, `dateTwoYearsAgo` и `date18YearsAgo`. После чего вступает в действие логика решений, сравнивая входящую переменную `dateOfBirth` с этими датами, и возвращает новый экземпляр нового класса: `Infant`, `Child` или `Adult`.

Использование класса Factory

Чтобы показать, насколько просто использовать класс `PersonFactory`, рассмотрим следующий код:

```
let factory = new PersonFactory();
let p1 = factory.getPerson(new Date(2017, 0, 20));
p1.printDetails();
let p2 = factory.getPerson(new Date(2010, 0, 20));
p2.printDetails();
let p3 = factory.getPerson(new Date(1969, 0, 20));
p3.printDetails();
```

Мы начинаем с создания переменной `personFactory` для хранения нового экземпляра класса `PersonFactory`. Затем мы создаем три переменные, `p1`, `p2` и `p3`, вызывая функцию `getPerson`, передавая три разные даты одной и той же функции. Вывод этого кода выглядит так:



Мы удовлетворили наши бизнес-требования и в то же время реализовали очень общий шаблон проектирования. Если мы оглянемся назад, то увидим, что у нас есть несколько четко определенных и простых классов. Классы `Infant`, `Child` и `Adult` имеют дело только с логикой, связанной с их классификацией и с тем, могут ли они подписывать контракты. Наш абстрактный базовый класс `Person` касается только логики, связанной с интерфейсом `IPerson`, а `PersonFactory` касается лишь логики, связанной с датой рождения. В этом примере показано, как объектно-ориентированные шаблоны проектирования и объектно-ориентированные свойства языка TypeScript могут помочь в написании качественного, расширяемого и обслуживаемого кода.

Резюме

В этой главе мы исследовали объектно-ориентированные концепции интерфейсов, классов и наследования. Мы обсудили как наследование интерфейсов, так и наследование классов и использовали полученные знания об интерфейсах, классах и наследовании для реализации шаблона проектирования Factory в TypeScript. В следующей главе мы рассмотрим расширенные возможности языка, включая обобщения, декораторы и методы асинхронных функций.

Глава 4

Декораторы, обобщения и асинхронные функции

Помимо концепций классов, интерфейсов и наследования, TypeScript использует ряд расширенных возможностей языка, чтобы помочь в разработке надежного объектно-ориентированного кода. Эти функции включают в себя декораторы, обобщения, промисы, а также использование ключевых слов `async` и `await` при работе с асинхронными функциями. Декораторы позволяют внедрять и запрашивать метаданные при работе с определениями классов, а также дают возможность программным путем подключаться к определению класса. Обобщения предоставляют способ написания процедур, когда точный тип используемого объекта неизвестен до времени выполнения. Промисы предоставляют возможность свободно писать асинхронный код, а функции `async await` приостанавливают выполнение до завершения асинхронной функции.

При написании крупномасштабных приложений на JavaScript эти языковые функции становятся частью инструментария программистов и позволяют применять многие шаблоны проектирования в коде JavaScript.

Эта глава разбита на три части. Первая часть посвящена декораторам, вторая – обобщениям, а третья – методам асинхронного программирования с использованием промисов и механизма `async await`. В более ранних версиях TypeScript промисы и async await могли использоваться только в ECMAScript 6 и выше, но теперь эта функциональность была расширена, чтобы включить цели ECMAScript 3.

Вот темы, которые мы рассмотрим в этой главе:

- синтаксис декораторов;
- фабрики декораторов;
- декораторы классов, методов и параметров;
- метаданные декоратора;
- синтаксис обобщений;
- использование и ограничение типа `T`;
- интерфейсы обобщений;
- промисы и синтаксис промисов;
- использование промисов;
- `async` и `await`;
- обработка ошибок `await`.

Декораторы

Декораторы в TypeScript предоставляют возможность подключаться к процессу определения класса программным путем. Помните, что определение класса описывает его форму. Другими словами, определение класса описывает, какие свойства есть у класса и какие методы он определяет. Только когда создается экземпляр класса, эти свойства и методы становятся доступными.

Декораторы, однако, позволяют внедрять код в фактическое определение класса. Декораторы могут использоваться в определениях классов, их свойствах и функциях и даже параметрах методов. Концепция декораторов существует в других языках программирования: в C# они называются атрибутами, а в Java – аннотациями.

В этом разделе мы рассмотрим, что такое декораторы, как их определяют и как их можно использовать. Мы рассмотрим декораторы классов, свойств, функций и методов.

Декораторы являются экспериментальным свойством компилятора TypeScript и были предложены как часть стандарта ECMAScript 7. TypeScript, однако, позволяет использовать декораторы в ES3 и выше. Чтобы использовать декораторы, в файл `tsconfig.json` в корне каталога вашего проекта должна быть добавлена новая опция компиляции. Эта опция называется `experimentalDecorators` и должна быть установлена со значением `true`:

```
{
   "compilerOptions": {
      "target": "es5",
      "module": "commonjs",
      "lib": [
        "es2015",
        "dom",
      ],
      "strict": true,
      "esModuleInterop": true,
      "experimentalDecorators": true,
      "emitDecoratorMetadata": true,
   }
}
```

Здесь мы указали, что опции компиляции будут использовать `es5` в качестве целевой платформы и что мы также используем опции `experimentalDecorators` и `emitDecoratorMetadata`. Обратите внимание, что мы также включили свойство `lib` и указали `es2015` и `dom` в качестве файлов библиотек, которые будет использовать компилятор. Мы будем экспериментировать с этими параметрами в следующей главе.

Синтаксис декораторов

Декоратор – это просто функция, которая вызывается с определенным набором параметров. Эти параметры автоматически заполняются средой выполнения JavaScript и содержат информацию о классе, к которому был применен декоратор. Количество параметров и типы этих параметров определяют, где можно применять декоратор. Чтобы проиллюстрировать синтаксис, используемый при работе с декораторами, давайте определим простой декоратор класса:

```
function simpleDecorator(constructor : Function) {
   console.log('simpleDecorator called.');
}
```

В данном случае мы определили функцию с именем `simpleDecorator`, которая принимает единственный параметр, `constructor` типа `Function`. Функция `simpleDecorator` записывает сообщение в консоль, указывая, что оно было вызвано. Эта функция является нашим определением декоратора. Чтобы использовать его, нам нужно применить его к определению класса:

```
@simpleDecorator
class ClassWithSimpleDecorator { }
```

Здесь мы применили декоратор к определению класса `ClassWithSimple Decorator`, используя символ «at» (`@`), за которым следует имя декоратора. Выполнение этого простого кода декоратора даст следующий результат:

```
simpleDecorator called.
```

Обратите внимание, что мы еще не создали экземпляр класса. Мы просто указали определение класса, добавили к нему декоратор, а наша функция-декоратор была вызвана автоматически. Это указывает на то, что декораторы применяются, когда класс определяется, а не когда создается его экземпляр. В качестве дополнительного примера рассмотрим следующий код:

```
let instance_1 = new ClassWithSimpleDecorator();
let instance_2 = new ClassWithSimpleDecorator();

console.log(`instance_1 : ${JSON.stringify(instance_1)}`);
console.log(`instance_2 : ${JSON.stringify(instance_2)}`);
```

Здесь мы создали два экземпляра класса `ClassWithSimpleDecorator` с именами `instance_1` и `instance_2`. Затем мы просто записываем сообщение в консоль. Вывод этого фрагмента кода выглядит так:

```
simpleDecorator called.
instance_1 : {}
instance_2 : {}
```

Этот вывод показывает, что функция-декоратор вызывается только один раз, независимо от того, сколько экземпляров одного и того же класса было создано или использовано. Декораторы вызываются только во время определения класса.

Несколько декораторов

К одной и той же цели можно применить несколько декораторов один за другим. В качестве примера рассмотрим следующий код:

```
function secondDecorator(constructor : Function) {
   console.log('secondDecorator called.')
}

@simpleDecorator
@secondDecorator
class ClassWithMultipleDecorators { }
```

Здесь мы создали еще один декоратор `secondDecorator`, который также просто записывает сообщение в консоль. Затем мы применяем `simpleDecorator` (из нашего предыдущего фрагмента кода) и декоратор `secondDecorator` к определению класса `ClassWithMultipleDecorators`. Вывод этого кода выглядит так:

```
secondDecorator called.
simpleDecorator called.
```

Тут мы видим один интересный момент, а именно: декораторы вызываются в обратном порядке их определения.

Декораторы оцениваются в порядке их появления в коде, но затем вызываются в обратном порядке.

Фабрика декораторов

Чтобы позволить декораторам принимать параметры, нам нужно использовать так называемую фабрику декораторов. Фабрика декораторов – это просто функция-обертка, которая возвращает саму функцию-декоратор. В качестве примера рассмотрим следующий код:

```
function decoratorFactory(name: string) {
   return function (constructor : Function ) {
      console.log(`decorator function called with : ${name}`);
   }
}
```

Здесь мы определили функцию с именем `decoratorFactory`, которая принимает один аргумент `name` типа `string`. Эта функция просто возвращает анонимную

функцию-декоратор, которая принимает единственный аргумент `constructor` типа `Function`. Анонимная функция (наша функция-декоратор) записывает значение параметра name в консоль.

Обратите внимание, что мы обернули функцию-декоратор в функцию `decoratorFactory`. Эта упаковка функции-декоратора – это то, что создает фабрику декораторов. Пока функция обертки возвращает функцию декоратора, это действительная фабрика декоратора.

Теперь мы можем использовать нашу фабрику декораторов:

```
@decoratorFactory('testName')
class ClassWithDecoratorFactory { }
```

Обратите внимание, что теперь мы можем передать параметр фабрике декораторов, как показано при использовании `@decoratorFactory` (‘`testName`’). Вывод этого кода выглядит так:

```
decorator function called with : testName
```

Есть несколько моментов, которые следует отметить касательно фабрик декораторов. Во-первых, сама функция-декоратор будет по-прежнему вызываться средой выполнения JavaScript с автоматически заполненными параметрами. Во-вторых, фабрика декораторов должна возвращать определение функции. Наконец, параметры, определенные для фабрики декораторов, могут использоваться внутри самой функции-декоратора, поскольку она находится в той же области действия, что и функция-декоратор.

Параметры декораторов класса

Все примеры, которые мы видели до сих пор, являются декораторами классов. Помните, что функция-декоратор будет автоматически вызываться средой выполнения JavaScript при объявлении класса. Наш декоратор функционирует до тех пор, пока все эти точки не будут определены, чтобы принимать один параметр с именем `constructor` типа `Function`. Среда выполнения JavaScript автоматически заполнит параметр `constructor`. Давайте рассмотрим этот параметр чуть более подробно.

Декораторы класса будут вызываться с помощью функции-конструктора декорированного класса. В качестве примера рассмотрим следующий код:

```
function classConstructorDec(constructor: Function) {
   console.log(`constructor : ${constructor}`);
}
@classConstructorDec
class ClassWithConstructor { }
```

Первой идет функция-декоратор с именем `classConstructorDec`, которая принимает один аргумент constructor типа `Function`. Первая строка этой функции просто выводит значение данного аргумента. Затем мы применяем этот декоратор к классу `ClassWithConstructor`. Вывод этого кода таков:

```
constructor : function ClassWithConstructor() { }
```

Таким образом, этот вывод показывает, что наша функция-декоратор вызывается с полным определением функции-конструктора класса, который она декорирует.

Давайте теперь обновим этот декоратор:

```
function classConstructorDec(constructor: Function) {
   console.log(`constructor : ${constructor}`);
   console.log(`constructor.name : ${(<any>constructor).name}`);
   constructor.prototype.testProperty = "testProperty_value";
}
```

Здесь мы обновили наш декоратор `classConstructorDec` двумя новыми строками кода. Первая строка выводит свойство name функции-конструктора в консоль. Обратите внимание, что мы должны были привести параметр `constructor` к типу `any`, чтобы успешно получить доступ к свойству `name`. Это необходимо, поскольку свойство функции name доступно только из ECMAScript 6 и только частично доступно в более ранних браузерах.

Последняя строка этой функции-декоратора на самом деле модифицирует прототип класса и добавляет свойство с именем `testProperty` (со значением `testProperty_value`) в само определение класса. Это пример того, как декораторы могут быть использованы для изменения определения класса. Затем мы можем получить доступ к этому свойству класса:

```
let classConstrInstance = new ClassWithConstructor();
console.log(`classConstrInstance.testProperty : `
   + `${(<any>classConstrInstance).testProperty}`);
```

Здесь мы создаем экземпляр класса `ClassWithConstructor` с именем `classConstrInstance`. Затем мы записываем значение свойства `testProperty` этого класса в консоль. Обратите внимание, что нам нужно привести тип переменной `classConstrInstance` к `any`, чтобы получить доступ к свойству `testProperty`. Это связано с тем, что мы не определили свойство `testProperty` в самом определении класса, а внедрили это свойство через декоратор. Вывод этого кода будет таким:

```
constructor : function ClassWithConstructor() {
}
constructor.name : ClassWithConstructor
classConstrInstance.testProperty : testProperty_value
```

Эти данные показывают, что мы можем использовать свойство name конструктора класса, чтобы найти имя самого класса. Мы также можем внедрить свойство класса с именем `testProperty` в определение класса. Как видно из вывода, значение свойства `testProperty` – `testProperty_value`, которое устанавливается в функции-декораторе.

Декораторы свойств

Декораторы свойств – это функции-декораторы, которые можно использовать в свойствах класса. Декоратор свойства вызывается с двумя параметрами – самим прототипом класса и именем свойства. В качестве примера рассмотрим следующий код:

```
function propertyDec(target: any, propertyKey : string) {
   console.log(`target : ${target}`);
   console.log(`target.constructor : ${target.constructor}`);
   console.log(`class name : `+ `${target.constructor.name}`);
   console.log(`propertyKey : ${propertyKey}`);
}

class ClassWithPropertyDec {
   @propertyDec
   name: string;
}
```

Здесь мы определили декоратор свойства с именем `propertyDec`, который принимает два параметра – `target` типа `any` и `propertyKey` типа `string`. В этом декораторе мы записываем ряд значений в консоль. Первое значение – это сам аргумент `target`. Второе значение, записанное в консоль, – свойство `constructor` объекта `target`. Третье значение – это имя функции-конструктора, а четвертое значение – сам аргумент `propertyKey`.

Затем мы определяем класс `ClassWithPropertyDec`, который теперь использует этот декоратор для свойства с именем `name`. Как и в случае с декораторами классов, синтаксис, используемый для декорирования свойства, – это просто наличие `@propertyDec` перед декорируемым свойством.

Вывод этого кода выглядит так:

```
target : [object Object]
target.constructor : function ClassWithPropertyDec() { }
target.constructor.name : ClassWithPropertyDec
propertyKey : name
```

Здесь видны результаты различных вызовов `console.log`. Аргумент `target` возвращает `[object Object]`, который просто указывает на то, что это про-

тотип объекта. Свойство `constructor` аргумента `target` возвращает функцию `ClassWithPropertyDec`, которая фактически является нашим конструктором класса. Свойство `name` этой функции-конструктора дает нам имя самого класса, а аргумент `propertyKey` – это имя самого свойства.

Поэтому декораторы свойств дают нам возможность проверить, было ли определенное свойство объявлено в экземпляре класса.

Декораторы статических свойств

Декораторы свойств также можно применять к свойствам статических классов. В нашем коде нет разницы в вызове синтаксиса – они такие же, как декораторы обычных свойств. Однако фактические аргументы, передаваемые во время выполнения, немного отличаются. Учитывая наше предыдущее определение свойства decorator `propertyDec`, рассмотрим, что происходит, когда этот декоратор применяется к свойству статического класса:

```
class StaticClassWithPropertyDec {
  @propertyDec
  static name: string;
}
```

Вывод различных функций `console.log` в нашем декораторе выглядит следующим образом:

```
target : function StaticClassWithPropertyDec() { }
target.constructor : function Function() { [native code] }
target.constructor.name : Function
propertyKey : name
```

Обратите внимание, что аргумент `target` (как показано в первой строке выходных данных) – это не прототип класса (как мы видели ранее), а фактическая функция-конструктор. Тогда определение `target.constructor` – это просто функция `Function`. `PropertyKey` остается неизменным, то есть `name`.

Это означает, что мы должны быть осторожнее с тем, что передается в качестве первого аргумента нашему декоратору свойств. Когда декорируемое свойство класса помечается как статическое, тогда целевым аргументом будет сам конструктор класса. Когда свойство класса не является статическим, целевой аргумент будет прототипом класса.

Давайте изменим наш декоратор свойств, чтобы правильно идентифицировать имя класса в обоих случаях:

```
function propertyDec(target: any, propertyKey : string) {
  if(typeof(target) === 'function') {
```

```
      console.log(`class name : ${target.name}`);
    } else {
      console.log(`class name : ${target.constructor.name}`);
    }
    console.log(`propertyKey : ${propertyKey}`);
}
```

Здесь вначале идет проверка природы аргумента `target`. Если вызов `typeof (target)` возвращает функцию, то мы знаем, что аргумент `target` является конструктором класса, и затем мы можем определить имя класса через `target.name`. Если вызов `typeof(target)` не возвращает `function`, мы знаем, что аргумент `target` – это прототип объекта и что вам нужно идентифицировать имя класса через свойство `target.constructor.name`.

Вывод этого кода выглядит так:

```
class name : ClassWithPropertyDec
propertyKey : name
class name : StaticClassWithPropertyDec
propertyKey : name
```

Наш декоратор свойств правильно идентифицирует имя класса, используется ли но в обычном свойстве класса или в статическом.

Декораторы методов

Декораторы методов – это декораторы, которые можно применять к методу класса. Декораторы методов вызываются средой выполнения JavaScript с тремя параметрами. Помните, что у декораторов классов есть только один параметр (прототип класса), а у декораторов свойств два параметра (прототип класса и имя свойства). У декораторов методов есть три параметра. Прототип класса, имя метода и (необязательно) дескриптор метода. Третий параметр, дескриптор метода, заполняется только при компиляции для ES5 и выше.

Давайте посмотрим на метод декоратора:

```
function methodDec (
   target: any,
   methodName: string,
   descriptor?: PropertyDescriptor)
{
   console.log(`target: ${target}`);
   console.log(`methodName : ${methodName}`);
   console.log(`target[methodName] : ${target[methodName]}`);
}
```

Здесь мы определили декоратор метода с именем `methodDec`, который принимает три наших параметра: `target`, `methodName` и `descriptor`. Обратите внимание, что свойство `descriptor` было помечено как необязательное. Первые две строки внутри декоратора просто записывают значения `target` и `methodName` в консоль. Однако обратите внимание на последнюю строку этого декоратора.

Здесь мы записываем значение `target[methodName]` в консоль. Таким образом, в консоль будет записано фактическое определение функции.

Теперь мы можем использовать этот декоратор метода в классе:

```
class ClassWithMethodDec {
   @methodDec
   print(output: string) {
      console.log(`ClassWithMethodDec.print`
         + `(${output}) called.`);
   }
}
```

Здесь мы определили класс с именем `ClassWithMethodDec`. У этого класса есть единственная функция `print`, которая принимает единственный параметр `output` типа `string`. Наша функция `print` просто записывает сообщение в консоль, включая значение аргумента `output`. Функция `print` была декорирована декоратором `methodDec`. Вывод этого кода выглядит так:

```
target: [object Object]
methodName : print
target[methodName] : function (output) {
   console.log("ClassWithMethodDec.print(" + output + ")
   called.");
}
```

Как видно из этого вывода, значение аргумента `target` – прототип класса. Значением аргумента `methodName` фактически является `print`. Результатом вызова `target[methodName]` является фактическое определение функции `target`.

Использование декораторов методов

Поскольку у нас есть определение функции, доступной нам в декораторе методов, мы могли бы использовать декоратор для изменения функциональности конкретной функции. Предположим, что мы хотим создать какой-то контрольный журнал и записывать сообщение в консоль каждый раз, когда вызывается метод. Это идеальный сценарий для декораторов методов.

Рассмотрим приведенный ниже код:

```
function auditLogDec(target: any, methodName: string
   descriptor?: PropertyDescriptor) {
   let originalFunction = target[methodName];
   let auditFunction = function (this: any) {
      console.log(`auditLogDec : overide of `
         ` ${methodName} called`);
      for (let i = 0; i < arguments.length; i++) {
         console.log(`arg : ${i} = ${arguments[i]}`);
      }
      originalFunction.apply(this, arguments);
   }
   target[methodName] = auditFunction;
   return target;
}
```

Здесь мы определили метод декоратора с именем `auditLogDec`. В этом декораторе мы создаем переменную `originalFunction` для хранения определения метода, который мы декорируем, так как `target[methodName]` вернет само определение функции. Затем мы создаем новую функцию `auditFunction` с единственным параметром `this` типа `any`. Использование параметра `this` в этом примере предназначено для работы с одним из строгих правил компилятора `noImplicitThis`. Мы обсудим эти правила в следующей главе.

Первая строка функции `auditFunction` записывает в консоль сообщение о том, что функция действительно была вызвана. Затем мы перебираем специальную переменную JavaScript с именем `arguments`, чтобы записать список аргументов в консоль. Обратите внимание, однако, на последнюю строку функции `auditFunction`. Мы используем JavaScript-функцию `apply` для вызова исходной функции, передавая параметры `this` и `arguments`.

После определения функции `auditFunction` мы назначаем ее функции исходного класса с помощью `target[methodName]`. По сути, мы заменили исходную функцию нашей новой `auditFunction` и в `auditFunction` вызвали оригинальную функцию через метод `apply`.

Чтобы увидеть это в действии, посмотрите на приведенное ниже объявление класса:

```
class ClassWithAuditDec {
   @auditLogDec
   print(arg1: string, arg2: string) {
      console.log(`ClassWithMethodDec.print`
         + `(${arg1}, ${arg2}) called.`);
   }
}
```

```
let auditClass = new ClassWithAuditDec();
auditClass.print("test1", "test2");
```

Здесь мы создаем класс с именем `ClassWithAuditDec` и декорируем функцию `print` с помощью нашего декоратора метода `auditLogDec`. У функции `print` есть два аргумента, и она просто записывает сообщение в консоль, показывая, что эта функция была вызвана с двумя аргументами. Последние две строки этого фрагмента кода являются примером того, как будет использоваться этот класс, и вывод будет таким:

```
auditLogDec : overide of print called
arg : 0 = test1
arg : 1 = test2
ClassWithMethodDec.print(test1, test2) called.
```

Как видно из этого вывода, наша функция-декоратор вызывается до фактической реализации функции `print`. Она также может записывать значения каждого аргумента в консоль, а затем правильно вызывает исходную функцию. Использование декораторов таким образом – это мощный метод незаметного внедрения дополнительной функциональности в объявление класса. Любой класс, который мы создаем, может легко включать в себя функциональные возможности аудита, просто декорируя соответствующие методы класса.

Декораторы параметров

Последний тип декоратора, который мы рассмотрим, – это декораторы параметров. Декораторы параметров используются для декорирования параметров конкретного метода. В качестве примера рассмотрим следующий код:

```
function parameterDec(
   target: any,
   methodName : string,
   parameterIndex: number)
   {
      console.log(`target: ${target}`);
      console.log(`methodName : ${methodName}`);
      console.log(`parameterIndex : ${parameterIndex}`);
   }
```

Здесь мы определили функцию с именем `parameterDec` с тремя аргументами. Аргумент `target` будет содержать прототип класса, как мы видели ранее. Аргумент `methodName` будет содержать имя метода, который содержит параметр, а аргумент `parameterIndex` будет содержать индекс параметра. Мы можем использовать эту функцию декоратора параметров следующим образом:

```
class ClassWithParamDec {
```

```
    print(@parameterDec value: string) {
    }
}
```

Здесь у нас есть класс `ClassWithParamDec`, который содержит единственную функцию `print`.

У этой функции `print` есть единственный аргумент `value` типа `string`. Мы декорировали параметр `value` декоратором `parameterDec`. Обратите внимание, что синтаксис для использования декоратора параметров (`@parameterDec`) такой же, как и любой другой декоратор. Вывод этого кода выглядит так:

```
target: [object Object]
methodName : print
parameterIndex : 0
```

Обратите внимание, что нам не дают никакой информации о параметре, который мы декорируем. Нам не говорят, какой это тип или его название. Поэтому декораторы параметров в действительности могут использоваться только для того, чтобы установить, что параметр был объявлен в методе.

Метаданные декораторов

Компилятор TypeScript также использует экспериментальную поддержку для того, что называется метаданными декораторов. Метаданные декораторов – это метаданные, которые генерируются в определениях классов, чтобы дополнять информацию, которая передается в декораторы. Эта опция называется `emitDecoratorMetadata` и может быть добавлена в файл `tsconfig.json` следующим образом:

```
{
    "compilerOptions": {
        // Другие опции
        "experimentalDecorators": true,
        "emitDecoratorMetadata": true
    }
}
```

После установки этой опции компилятор TypeScript будет генерировать дополнительную информацию, касающуюся наших определений классов. Чтобы увидеть результаты этой опции компиляции, нужно поближе взглянуть на сгенерированный код JavaScript. Рассмотрим следующий параметр декоратора и определение класса:

```
function metadataParameterDec(target: any,
    methodName : string, parameterIndex: number) { }
```

```
class ClassWithMetaData {
   print(
      @metadataParameterDec
      id: number,
      name: string) : number { return 1000; }
}
```

Здесь у нас есть стандартный декоратор параметров `metadataParameterDec` и определение класса `ClassWithMetaData`. Мы декорируем первый параметр функции `print`. Если мы не используем опцию компиляции `emitDecoratorMetadata` или если для нее установлено значение `false`, наш сгенерированный код JavaScript будет определен следующим образом:

```
var ClassWithMetaData = (function () {
   function ClassWithMetaData() {
   }
   ClassWithMetaData.prototype.print = function (id, name) { };
   __decorate([
      __param(0, metadataParameterDec)
   ], ClassWithMetaData.prototype, "print");
   return ClassWithMetaData;
}());
```

Этот сгенерированный код JavaScript определяет стандартное замыкание JavaScript для нашего класса `ClassWithMetaData`. Интересующий нас код находится в нижней части замыкания, где компилятор TypeScript внедрил метод `_decorate`. Мы не будем заботиться о полной функциональности этого метода, за исключением того, что обратим внимание на следующий факт: она содержит информацию о функции `print` и указывает на то, что она называется `print` и что у нее есть единственный параметр с индексом `0`.

Если для опции `emitDecoratorMetadata` установлено значение `true`, сгенерированный код JavaScript будет содержать дополнительную информацию о функции `print`:

```
var ClassWithMetaData = (function () {
   function ClassWithMetaData() { }
   ClassWithMetaData.prototype.print = function (id, name) { };
   __decorate([
      __param(0, metadataParameterDec),
      __metadata('design:type', Function),
      __metadata('design:paramtypes', [Number, String]),
      __metadata('design:returntype', Number)
   ], ClassWithMetaData.prototype, "print");
   return ClassWithMetaData;
}());
```

Обратите внимание, что функция `__decorate` теперь включает в себя дополнительные вызовы функции `__metadata`, которая вызывается три раза. Первый вызов использует специальный ключ метаданных `'design:type'`, второй использует ключ `'design:paramtypes'`, а третий вызов – ключ `'design:returntype'`. Эти три вызова функции `__metadata` фактически регистрирует дополнительную информацию о самой функции print. Ключ `'design:type'` используется для регистрации факта, что функция `print` имеет тип `Function`. Ключ `'design:paramtypes'` используется для регистрации того факта, что у функции `print` есть два параметра: первый – `Number`, а второй – `String`. Ключ `'design:returntype'` используется для регистрации возвращаемого типа функции `print`, которое в нашем случае является числом.

Использование метаданных декораторов

Чтобы использовать эту дополнительную информацию в декораторе, нам понадобится сторонняя библиотека `reflect-metadata`. Мы подробно обсудим, как использовать сторонние библиотеки, в последующих главах, но пока эту библиотеку можно включить в наш проект, набрав из командной строки следующее:

```
npm install reflect-metadata --save-dev
```

После того как она будет установлена, нам нужно будет обратиться к ней в нашем файле TypeScript, добавив следующую строку в верхней части файла:

```
import 'reflect-metadata';
```

Прежде чем мы попытаемся скомпилировать любой код, использующий библиотеку `reflect-metadata`, нам нужно установить файл объявления для этой библиотеки:

```
npm install @types/reflect-metadata --save-dev
```

Мы подробно обсудим файлы объявлений в следующей главе.

Теперь мы можем приступить к использованию метаданных этого класса, вызвав функцию `Reflect.getMetadata` в нашем декораторе. Рассмотрим обновленный вариант предыдущего декоратора параметров:

```
function metadataParameterDec(target: any,
   methodName : string,
   parameterIndex: number) {
      let designType = Reflect.getMetadata(
        "design:type", target, methodName);
        console.log(`designType: ${designType}`)
      let designParamTypes = Reflect.getMetadata(
        "design:paramtypes", target, methodName);
        console.log(`paramtypes : ${designParamTypes}`);
```

```
        let designReturnType = Reflect.getMetadata(
          "design:returntype", target, methodName);
          console.log(`returntypes : ${designReturnType}`);
    }
```

Здесь мы обновили наш декоратор параметров тремя вызовами функции `Reflect.getMetadata`. Первый использует ключ метаданных `'design:type'`. Это тот же ключ метаданных, который мы видели ранее в сгенерированном коде JavaScript, где компилятор генерировал вызовы функции `__metadata`. Затем мы записываем результат в консоль, а потом повторяем этот процесс для ключей `'design:paramtypes'` и `'design:returntype'`. Вывод этого кода выглядит так:

```
designType: function Function() { [native code] }
paramtypes : function Number() { [native code] },function
String() { [native code] }
returntypes : function Number() { [native code] }
```

Видно, что по своей природе функция `print` (как записано ключом метаданных `'design: type'`) – это `Function`. Также видно, что информация, возвращаемая ключом `'design: paramtypes'`, – это массив, который включает в себя `Number` и `String`. Следовательно, этот массив указывает на то, что у функции есть два параметра: первый типа `Number` и второй типа `String`. Наконец, возвращаемый тип этой функции – число.

Метаданные, которые автоматически генерируются компилятором TypeScript и могут быть прочитаны и опрошены во время выполнения, могут быть чрезвычайно полезными. В других языках, таких как C#, этот тип метаданных называется отражением и является фундаментальным принципом при написании фреймворков для внедрения зависимостей или для генерации инструментов анализа кода.

Обобщения

Обобщения – это способ написания кода, который будет иметь дело с любым типом объекта, но при этом сохранит целостность данного типа. До сих пор мы использовали интерфейсы, классы и базовые типы TypeScript для обеспечения строго типизированного (и менее подверженного ошибкам) кода в наших примерах. Но что происходит, если блок кода должен работать с любым типом объекта?

В качестве примера предположим, что мы хотели написать некий код, который мог бы перебирать массив объектов и возвращать конкатенацию их значений. Итак, учитывая список чисел, скажем `[1,2,3]`, он должен вернуть строку `1,2,3`. Или, учитывая список строк, скажем `[first, second, third]`, он вернет строку `first,second,third`. Мы могли бы написать код, который бы принимал значения

типа any, но это может привести к ошибкам – помните S.F.I.A.T.? Мы хотим убедиться, что все элементы массива имеют одинаковый тип. Вот тут в игру вступают обобщения.

Синтаксис обобщений

В качестве примера синтаксиса обобщений TypeScript давайте напишем класс с именем `Concatenator`, который выполнит конкатенацию значений в массиве. Нам нужно убедиться, что каждый элемент массива имеет одинаковый тип. Этот класс должен уметь обрабатывать массивы строк, массивы чисел и, по сути, массивы любого типа. Для этого нам нужно полагаться на функциональность, которая является общей для каждого из этих типов. Поскольку у всех объектов JavaScript есть функция `toString` (которая вызывается всякий раз, когда во время выполнения требуется строка), мы можем использовать эту функцию для создания обобщенного класса, который выводит все значения, содержащиеся в массиве.

Обобщенная реализация класса Concatenator выглядит так:

```
class Concatenator< T > {
   concatenateArray(inputArray: Array< T >): string {
      let returnString = "";
      for (let i = 0; i < inputArray.length; i++) {
         if (i > 0) returnString += ",";
         returnString += inputArray[i].toString();
      }
      return returnString;
   }
}
```

Первое, что мы замечаем, – это синтаксис объявления класса `Concatenator <T>`. `<T>` – это синтаксис, используемый для обозначения обобщенного типа, а имя, используемое для этого типа в остальной части нашего кода, – `T`. Функция `concatenateArray` также использует синтаксис обобщенного типа, `Array <T>`. Это указывает на то, что аргумент `inputArray` должен быть массивом типа, который первоначально использовался для создания экземпляра этого класса.

Создание экземпляра обобщенного класса

Чтобы использовать экземпляр этого обобщенного класса, нам нужно сконструировать класс и сообщить компилятору через синтаксис `<>`, что такое фактический тип `T`. Мы можем использовать любой тип для типа `T` в этом синтаксисе, включая базовые типы, классы или даже интерфейсы. Давайте создадим несколько версий этого класса:

```
var stringConcat = new Concatenator<string>();
var numberConcat = new Concatenator<number>();
```

Обратите внимание на синтаксис, который мы использовали для создания экземпляра класса `Concatenator`. В первой строке этого примера мы создаем экземпляр обобщенного класса `Concatenator` и указываем, что он должен заменять обобщенный тип `T` на тип `string` везде, где используется `T`. Аналогично, во второй строке создается экземпляр класса `Concatenator` и указывается, что тип number должен использоваться везде, где встречается обобщенный тип `T`.

Если мы используем этот простой принцип подстановки, то в случае с экземпляром `stringConcat` (который использует строки) аргумент `inputArray` должен иметь тип `Array<string>`. Аналогично, экземпляр `numberConcat` этого обобщенного класса использует числа, поэтому аргумент `inputArray` должен быть массивом чисел. Чтобы проверить эту теорию, давайте сгенерируем массив строк и массив чисел и посмотрим, что скажет компилятор, если мы попытаемся нарушить это правило:

```
var stringArray: string[] = ["first", "second", "third"];
var numberArray: number[] = [1, 2, 3];
var stringResult = stringConcat.concatenateArray(stringArray);
var numberResult = numberConcat.concatenateArray(numberArray);
var stringResult2 = stringConcat.concatenateArray(numberArray);
var numberResult2 = numberConcat.concatenateArray(stringArray);
```

Наши первые две строки определяют переменные `stringArray` и `numberArray` для хранения соответствующих массивов. Затем мы передаем переменную `stringArray` в функцию `stringConcat` – никаких проблем. На следующей строке мы передаем `numberArray` в `numberConcat` – пока порядок.

Однако проблемы начинаются, когда мы пытаемся передать массив чисел экземпляру `stringConcat`, который был настроен только на использование строк. Опять же, если мы попытаемся передать массив строк экземпляру `numberConcat`, который был настроен таким образом, чтобы разрешать только числа, TypeScript выдаст ошибки:

```
error TS2345: Argument of type 'number[]' is not assignable
to parameter of type 'string[]'.
Type 'number' is not assignable to type 'string'.
error TS2345: Argument of type 'string[]' is not assignable
to parameter of type 'number[]'.
Type 'string' is not assignable to type 'number'.
```

Ясно, что мы пытаемся передать массив чисел, где мы должны были использовать строки, и наоборот. Опять же, компилятор предупреждает нас, что мы используем код неправильно, и вынуждает нас решить эти проблемы, прежде чем продолжить.

Эти ограничения в обобщениях являются свойством TypeScript, относящимся к категории compile-time only. Если мы посмотрим на сгенерированный код JavaScript, то не увидим груды кода, которые должны выполнять все требования, чтобы эти правила переносились в результирующий JavaScript. Все эти ограничения типов и синтаксис обобщений фактически скомпилированы. В случае с обобщениями сгенерированный код JavaScript на самом деле является очень упрощенной версией нашего кода, без видимого типа.

Использование типа T

При использовании обобщений важно отметить, что весь код в определении обобщенного класса или обобщенной функции должен учитывать свойства T, как если бы это был тип объекта. Давайте подробнее рассмотрим реализацию функции `concatenateArray` в этом свете:

```
class Concatenator< T > {
   concatenateArray(inputArray: Array< T >): string {
      let returnString = "";
      for (let i = 0; i < inputArray.length; i++) {
         if (i > 0) returnString += ",";
         returnString += inputArray[i].toString();
      }
      return returnString;
   }
}
```

Функция `concatenateArray` строго типизирует аргумент `inputArray`, поэтому он должен иметь тип `Array<T>`. Это означает, что любой код, использующий аргумент `inputArray`, может применять только те функции и свойства, которые являются общими для всех массивов, независимо от типа массива. Мы использовали `inputArray` в двух местах.

Во-первых, в объявлении цикла `for` обратите внимание на то, где мы использовали свойство `inputArray.length`. У всех массивов есть свойство `length` для указания количества элементов в массиве, поэтому использование `inputArray.length` будет работать с любым массивом, независимо от типа объекта, который содержит массив. Во-вторых, в последней строке цикла `for` мы обратились к объекту в массиве, когда использовали синтаксис `inputArray [i]`.

Эта ссылка фактически возвращает нам единственный объект типа `T`. Помните, что всякий раз, когда мы используем `T` в нашем коде, мы должны использовать только те функции и свойства, которые являются общими для любого объекта типа `T`. К счастью для нас, мы используем только функцию `toString`, а у всех объектов JavaScript, независимо от их типа, есть действительная функция `toString`.

Так что этот блок кода будет скомпилирован чисто.

Давайте проверим теорию типа T, создав собственный класс для передачи в класс Concatenator:

```
class MyClass {private _name: string;
   constructor(arg1: number) {
      this._name = arg1 + "_MyClass";
   }
}
```

Здесь у нас идет простое определение класса с именем MyClass с функцией-конструктором, которая принимает число. Эта функция устанавливает имя внутренней переменной _name в значение аргумента arg1.

Давайте теперь создадим массив экземпляров MyClass:

```
let myArray: MyClass[] = [
   new MyClass(1),
   new MyClass(2),
   new MyClass(3)];
```

Теперь мы можем создать экземпляр нашего обобщенного класса Concatenator:

```
let myArrayConcatentator = new Concatenator<MyClass>();
let myArrayResult =
   myArrayConcatentator.oncatenateArray(myArray);
console.log(myArrayResult);
```

Первым идет экземпляр класса Concatenator, указывая, что он будет работать только с объектами типа MyClass. Затем мы вызываем функцию concatenateArray и сохраняем результат в переменной с именем myArrayResult. Наконец, мы выводим результат на консоль. После запуска этого кода будет выведено следующее:

```
[object Object],[object Object],[object Object]
```

Не совсем то, что мы ожидали. Этот вывод связан с тем, что строковое представление объекта, то есть не относящегося к основным типам JavaScript, возвращает [object type]. Любой пользовательский объект, который вы пишете, должен переопределить функцию toString, чтобы обеспечить читабельный вывод. Мы можем легко исправить этот код, предоставив переопределение функции toString в нашем классе:

```
class MyClass {
   private _name: string;
   constructor(arg1: number) {
      this._name = arg1 + "_MyClass";
```

```
    }
    toString(): string {
      return this._name;
    }
}
```

Здесь мы заменили стандартную функцию `toString`, которую наследуют все объекты JavaScript, своей собственной реализацией. В этой функции мы просто вернули значение закрытой переменной `_name`. При запуске этого примера мы теперь получим ожидаемый результат:

```
1_MyClass,2_MyClass,3_MyClass
```

Ограничение типа T

При использовании обобщений часто желательно ограничивать тип `T` только конкретным типом или подмножеством типов. В этих случаях мы не хотим, чтобы наш обобщенный код был доступен для любого типа объекта; мы хотим, чтобы он был доступен только для определенного подмножества объектов. TypeScript использует наследование, чтобы добиться этого с помощью обобщений.

В качестве примера давайте определим интерфейс для футбольной команды:

```
enum ClubHomeCountry {
   England,
   Germany
}

interface IFootballClub {
   getName() : string | undefined;
   getHomeCountry(): ClubHomeCountry | undefined;
}
```

Здесь мы определили перечисление с именем `ClubHomeCountry`, чтобы указать, где находится родная страна футбольного клуба. Затем наш интерфейс `IFootballClub` определяет два метода, которые должен реализовать любой класс `FootballClub`: функцию `getName`, которая возвращает имя футбольного клуба, и функцию `getHomeCountry`, которая возвращает значение перечисления `ClubHomeCountry`.

Теперь мы определим абстрактный базовый класс для реализации этого интерфейса:

```
abstract class FootballClub implements IFootballClub {
   protected _name: string | undefined;
   protected _homeCountry: ClubHomeCountry | undefined;
   getName() { return this._name };
```

```
    getHomeCountry() { return this._homeCountry };
}
```

Этот абстрактный класс с именем `FootballClub` реализует интерфейс `IFootballClub`, определяя функции `getName` и `getHomeCountry`, которые возвращают значения, хранящиеся в защищенных переменных `_name` и `_homeCountry` соответственно. Поскольку это абстрактный класс, нам нужно наследовать от него, чтобы создать конкретные классы:

```
class Liverpool extends FootballClub {
    constructor() {
        super();
        this._name = "Liverpool F.C.";
        this._homeCountry = ClubHomeCountry.England;
    }
}

class BorussiaDortmund extends FootballClub {
    constructor() {
        super();
        this._name = "Borussia Dortmund";
        this._homeCountry = ClubHomeCountry.Germany;
    }
}
```

Здесь мы определили два класса, названных `Liverpool` и `BorussiaDortmund`, полученных в результате наследования от нашего абстрактного базового класса `FootballClub`. У обоих классов есть только конструктор, который устанавливает внутренние свойства `_name` и `_homeCountry`.

Теперь мы можем создать обобщенный класс, который будет работать с любым классом, реализующим интерфейс `IFootballClub`:

```
class FootballClubPrinter< T extends IFootballClub >
    implements IFootballClubPrinter< T > {
    print(arg : T) {
        console.log(` ${arg.getName()} is ` +
            `${this.IsEnglishTeam(arg)}` +
            ` an English football team.`
        );
    }
    IsEnglishTeam(arg : T) : string {
        if ( arg.getHomeCountry() == ClubHomeCountry.England )
            return "";
        else return "NOT"
    }
}
```

Здесь мы определили класс с именем `FootballClubPrinter`, который использует синтаксис обобщений. Обратите внимание, что обобщенный тип `T` теперь является производным от интерфейса `IFootballClub`, на что указывает ключевое слово `extends` в `<T extends IFootballClub>`. Использование наследования при определении обобщенного класса ограничивает тип класса, который может использоваться этим обобщенным кодом. Другими словами, любое использование типа `T` в обобщенном коде будет вместо этого заменять интерфейс `IFooballClub`. Это означает, что обобщенный код разрешает использовать только функции или свойства, которые определены в интерфейсе `IFootballClub`, везде, где используется `T`.

Эти ограничения можно увидеть в функции print, а также в функции `IsEnglishTeam`. В функции `print` аргумент `arg` имеет тип `T`, и поэтому мы можем использовать `arg.getName`, который был определен в интерфейсе `IFootballClub`. В функции `print` мы также вызываем функцию `IsEnglishTeam` и передаем ей аргумент `arg`. Функция `IsEnglishTeam` использует функцию `getHomeCountry`, которая также определена в интерфейсе `IFootballClub`.

Чтобы проиллюстрировать, как можно применять обобщенный класс `FootballClubPrinter`, рассмотрим приведенный ниже код:

```
let clubInfo = new FootballClubPrinter();
clubInfo.print(new Liverpool());
clubInfo.print(new BorussiaDortmund());
```

Здесь мы создаем экземпляр класса `FootballClubPrinter` с именем `clubInfo`.

Обратите внимание, что нам не нужно указывать тип (как мы делали в случае с предыдущим классом `Concatenator`) при создании экземпляра класса.

Затем мы вызываем функцию `print` обобщенного класса `clubInfo`, передавая новый экземпляр класса `Liverpool`, а потом экземпляр класса `BorussiaDortmund`. Вывод этого кода выглядит так:

```
Liverpool F.C. is an English football team.
Borussia Dortmund is NOT an English football team.
```

Таким образом, мы можем ограничить тип `T`, который используется в обобщенном синтаксисе, конкретным типом, выведя тип `T` из другого типа, используя синтаксис extends. Другими словами, если `T` расширяет тип, то обобщенный код может предполагать, что `T` относится к этому типу.

Обобщенные интерфейсы

Мы также можем использовать интерфейсы с синтаксисом обобщенного типа. Если бы мы создали интерфейс для обобщенного класса `FootballClubPrinter`, определение интерфейса выглядело бы так:

```
interface IFootballClubPrinter < T extends IFootballClub > {
  print(arg : T);
  IsEnglishTeam(arg : T);
}
```

Этот интерфейс выглядит идентично нашему определению класса, с той лишь разницей, что функции `print` и `IsEnglishTeam` не имеют реализации. Мы сохранили синтаксис обобщенного типа, используя <T>, и дополнительно указали, что тип `T` должен реализовывать интерфейс `IFootballClub`. Чтобы использовать этот интерфейс с классом `FootballClubPrinter`, мы можем изменить определение класса:

```
class FootballClubPrinter< T extends IFootballClub >
  implements IFootballClubPrinter<T> { }
```

Этот синтаксис кажется довольно простым. Как мы уже видели, мы используем ключевое слово `implements` после определения класса, а затем используем имя интерфейса. Однако обратите внимание, что мы передаем тип `T` в определение интерфейса `IFootballClubPrinter` как обобщенный тип `IFootballClubPrinter<T>`. Это удовлетворяет определению обобщенного интерфейса `IFootballClubPrinter`.

Интерфейс, который определяет наши обобщенные классы, дополнительно защищает наш код от непреднамеренного изменения. В качестве примера предположим, что мы попытались переопределить определение класса `FootballClubPrinter`, так что `T` не ограничен типом `IFootballClub`:

```
class FootballClubPrinter<T>
  implements IFootballClubPrinter<T> { }
```

Здесь мы удалили ограничение для типа `T` для класса `FootballClubPrinter`.

TypeScript автоматически выдаст ошибку:

```
error TS2344: Type 'T' does not satisfy the constraint
'IFootballClub'.
```

Эта ошибка указывает на ошибочное определение класса; тип `T`, используемый в коде (`FootballClubPrinter<T>`), должен использовать тип `T`, который наследует от `IFootballClub`, чтобы правильно реализовать интерфейс `IFootballClubPrinter`.

Создание новых объектов в обобщениях

Время от времени обобщенным классам может понадобиться создать объект типа, который передается как обобщенный тип `T`. Рассмотрим следующий код:

```
class FirstClass {
   id: number | undefined; }

class SecondClass {
   name: string | undefined; }

class GenericCreator< T > {
   create(): T {
      return new T();
   }
}

var creator1 = new GenericCreator<FirstClass>();
var firstClass: FirstClass = creator1.create();
var creator2 = new GenericCreator<SecondClass>();
var secondClass : SecondClass = creator2.create();
```

Здесь у нас есть два определения классов, `FirstClass` и `SecondClass`. У `FirstClass` есть открытое свойство `id`, а у `SecondClass` – открытое свойство `name`. Далее следует обобщенный класс, который принимает тип `T`, и у него есть единственная функция с именем `create`. Эта функция пытается создать новый экземпляр класса типа `T`.

Последние четыре строки данного примера показывают нам, как мы хотели бы использовать этот обобщенный класс. Переменная `creator1` создает новый экземпляр класса `GenericCreator`, используя правильный синтаксис для создания переменных типа `FirstClass`. Переменная `creator2` – это новый экземпляр класса `GenericCreator`, но на этот раз используется `SecondClass`. К сожалению, предыдущий код выдаст ошибку компиляции:

```
error TS2304: Cannot find name 'T'.
```

Согласно документации TypeScript, чтобы дать возможность обобщенному классу создавать объекты типа `T`, нам нужно обратиться к типу `T` посредством его функции-конструктора. Нам также нужно передать определение класса в качестве аргумента. Функцию `create` нужно переписать:

```
class GenericCreator< T > {
   create(arg1: { new(): T }) : T {
      return new arg1();
   }
}
```

Давайте разберем эту функцию по частям. Сначала мы передаем аргумент с именем `arg1`. Затем этот аргумент определяется как тип `{new(): T}`.

Это маленькая хитрость, которая позволяет нам обращаться к `T` с помощью функции-конструктора. Мы определяем новый анонимный тип, который перегружает

функцию `new()` и возвращает тип `T`. Это означает, что аргумент `arg1` – это строго типизированная функция, у которой есть единственный конструктор, который возвращает тип `T`. Реализация этой функции просто возвращает новый экземпляр переменной `arg1`. Использование этого синтаксиса устраняет ошибку компиляции, с которой мы столкнулись ранее.

Это изменение, однако, означает, что мы должны передать определение класса функции `create`:

```
var creator1 = new GenericCreator<FirstClass>();
var firstClass: FirstClass = creator1.create(FirstClass);

var creator2 = new GenericCreator<SecondClass>();
var secondClass : SecondClass = creator2.create(SecondClass);
```

Обратите внимание на изменение при использовании функции create. Теперь мы должны передать определение класса для нашего типа `T` – `create (FirstClass)` и `create (SecondClass)` в качестве первого аргумента. Попробуйте запустить этот код, чтобы увидеть, что происходит. Обобщенный класс, по сути, будет создавать новые объекты типов `FirstClass` и `SecondClass`, как и ожидалось.

Расширенные типы с обобщениями

Компилятор TypeScript уже предоставляет нам большой набор инструментов для определения пользовательских типов, наследования типов и использования синтаксиса обобщений для описания того, как наш код намеревается использовать эти типы. Сочетая эти языковые особенности, мы можем приступить к описанию определений по-настоящему расширенных типов, включая типы, основанные на других типах, или типы, основанные на некоторых или всех свойствах другого типа. Используя новые ключевые слова TypeScript и применяя некоторые общеизвестные языковые свойства, мы также можем начать запрашивать тип для его свойства, проверить, есть ли у типа определенные свойства, или полностью изменить тип, добавив либо удалив свойства по своему усмотрению. Добро пожаловать в сногсшибательный мир условных типов, выводимых типов, отображаемых типов и т. д., или, как его описывает автор, просто в «математику теоретических типов». Имейте в виду, что синтаксис расширенных типов и обобщений на первый взгляд может показаться сложным, но если мы применим некоторые простые правила, он в конечном итоге станет понятным.

Помните, что хотя типы помогают нам описывать наш код, а также помогают укрепить его, они не влияют на генерируемый JavaScript. Простое описание типа является теоретическим упражнением, и большая часть математики расширенных типов, которую мы будем изучать, только генерирует тип. Нам все еще нужно будет использовать эти типы, чтобы реализовать их преимущества на практике.

Условные типы

Одна из особенностей языка TypeScript включает в себя простую, оптимизированную версию оператора `if then else`, которая использует символ вопросительного знака (?) для определения оператора `if` и символ двоеточия (:) для определения пути `then` и `else`. Они называются условными операторами. Формат условного оператора выглядит так:

```
(conditional statement) ? (true value) : (false value);
```

Вначале идет условный оператор, за которым следует символ ?. Затем мы определяем значения `true` и `false`, разделенные символом :. В качестве примера рассмотрим следующий код:

```
let trueValue = true;
let printValue = trueValue === true ? "true" : "false";
console.log(`printValue is : ${printValue}`);
```

Здесь мы определили переменную с именем `trueValue` и установили для нее значение `true`. Затем наш условный оператор устанавливает значение переменной `printValue` в строку `true` или `false`, основываясь на проверке условия `if trueValue === true`. Другими словами, если `trueValue === true`, используйте строковое значение `true`, в противном случае используйте значение после символа двоеточия (:), которое является строковым значением `false`.

Условные операторы также могут использоваться с типами, использующими тот же синтаксис, а результирующие типы называются условными. Рассмотрим следующее определение типа:

```
type numberOrString<T> = T extends number ? number : string;
```

Здесь мы определили новый тип `numberOrString`, который использует синтаксис обобщений, чтобы указать, что он может применяться только при заданном типе `T`. Если тип `T` расширяет число, то результирующий тип `numberOrString` будет иметь тип `number`. Если тип `T` не расширяет число, то результирующий тип будет иметь тип `string`.

Затем мы можем использовать этот условный тип в функции:

```
function isNumberOrString<T>(input: numberOrString<T>) {
   console.log(`numberOrString : ${input}`); }
```

Здесь у нас есть функция с именем `isNumberOrString`, которая использует синтаксис обобщений и, следовательно, требует, чтобы мы вызывали эту функцию с типом `T`. Единственный параметр этой функции – `input`, который использует наш условный тип `numberOrString<T>`.

Это означает, что в зависимости от типа T, с которым мы вызываем эту функцию, входной параметр может измениться с числа на строку. Другими словами, параметр input является условным для типа T. Попробуем вызвать эту функцию:

```
isNumberOrString<number>(1);
isNumberOrString<number>("test");
```

Вначале мы вызываем функцию isNumberOrString, указывая, что тип T является числом, и передавая аргумент 1. Затем мы вызываем ту же функцию, с тем же типом T, с аргументом "test". Компиляция этого кода вызовет ошибку:

```
error TS2345: Argument of type '"test"' is not assignable to
parameter of type 'number'.
```

Так что же здесь происходит? Мы вызываем функцию isNumberOrString, передавая тип number для универсального типа T, а затем передавая аргумент типа string.

Наш условный тип numberOrString<T> в данном случае является ключом. Возвращаясь к определению этого условного типа, мы знаем, что если тип T является числом, он возвращает тип number. Если тип T не является числом, он возвращает тип string. Поэтому, когда мы вызываем функцию isNumberOrString и указываем, что тип T является числом, параметр input меняет свой тип на строку. Это означает, что для исправления нашей ошибки компиляции нам нужно вызвать эту функцию следующим образом:

```
isNumberOrString<string>("test");
```

Здесь тип T имеет тип string и, следовательно, входной параметр изменил свой тип на string, поэтому мы можем вызвать эту функцию с аргументом input типа string.

Давайте теперь попробуем более сложный пример. У нас есть следующие типы с именами a, ab и abc:

```
interface a {
   a: number;
}

interface ab {
   a: number;
   b: string;
}

interface abc {
   a: number;
   b: string;
   c: boolean;
}
```

Здесь у нас есть три типа. У типа a есть единственное свойство с именем a, у типа ab два свойства – a и b, а у типа abc три свойства – a, b и c. Теперь мы можем создать условный тип:

```
type abc_ab_a<T> = T extends abc ? [number, string, boolean] :
   T extends ab ? [number, string] :
   T extends a ? [number]
   : never;
```

Здесь мы определили условный тип с именем abc_ab_a. Если тип T расширяет abc (другими словами, если у него есть свойства a, b и c), тогда возвращается кортеж [number, string, boolean]. Если тип T не расширяет abc, мы проверяем, расширяет ли он ab; другими словами, есть ли у него свойства a и b. Если да, то возвращается кортеж [number, string]. Если тип T не расширяет ab, мы проверяем, расширяет ли он a, и возвращаем кортеж [number]. Если тип T не расширяет abc, ab или a, тогда возвращается never.

Затем мы можем написать функцию, которая использует этот условный тип:

```
function getKeyAbc<T>(key: abc_ab_a<T>): string {
   let [...args] = key;
   let keyString = ":";
   for (let arg of args) {
      keyString += `${arg}:`
   }
   return keyString;
}
```

Наша функция называется getKeyAbc и использует синтаксис обобщений, чтобы указать, что она должна вызываться с типом T. У этой функции есть единственный аргумент key, который использует наш условный тип abc_ab_a. Помните, что наш условный тип вернет кортеж, поэтому key – это на самом деле кортеж, у которого может быть три, два или один элемент. Тело этой функции использует spread-кортеж для извлечения значений аргумента key и объединения их в строку. Мы можем использовать эту функцию следующим образом:

```
let key10 = getKeyAbc<a>([1]);
console.log(`key10 : ${key10}`);

let key20 = getKeyAbc<ab>([1, "test"]);
console.log(`key20 : ${key20}`);

let key30 = getKeyAbc<abc>([1, "test2", true]);
console.log(`key30 : ${key30}`);
```

Вначале идет переменная key10, которая вызывает нашу функцию getKeyAbc с типом a. Обратите внимание, что поскольку мы используем тип a, условный тип abc_ab_a вернет кортеж типа [number], поэтому входной параметр функ-

ции `getKeyAbc` должен быть кортежем типа `[number]`. Затем мы выводим значение возвращаемой строки в консоль. Переменная `key20` вызывает функцию `getKeyAbc` с типом `ab` и поэтому должна предоставлять входной аргумент типа `[number, string]`. Переменная `key30` использует тип `abc` и поэтому должна предоставлять входной аргумент типа `[number, string, boolean]`.

Вывод этого кода выглядит так:

```
key10 : :1:
key20 : :1:test:
key30 : :1:test2:true:
```

Все хорошо. Давайте теперь попробуем немного нарушить правила:

```
let keyNever = getKeyAbc<string>([1]);
```

Здесь мы передаем строку как тип `T` для нашей функции `getKeyAbc`. Этот код выдаст ошибку компиляции:

```
error TS2345: Argument of type '"test"' is not assignable to
parameter of type 'never'.
```

Причина данной ошибки состоит в том, что мы использовали тип `string` для типа `T`. Возвращаясь к нашему условному типу `abc_ab_a`, мы видим, что строка не расширяет ни `a`, `ab`, ни `abc`, и поэтому условный тип возвращает тип `never`. Это гарантирует, что функция `getKeyAbc` должна использоваться только с одним из трех разрешенных типов.

Если у нас есть несоответствие типа `T` и входного параметра:

```
let keyABCWrong = getKeyAbc<abc>([1, "test"]);
```

компилятор выдаст следующее сообщение:

```
error TS2345: Argument of type '[number, string]' is not
assignable to parameter of type '[number, string, boolean]'.
Property '2' is missing in type '[number, string]'
```

Здесь мы указали, что тип `T` имеет тип `abc`, и, следовательно, входной параметр должен быть кортежем типа `[number, string, boolean]`.

Распределенные условные типы

При определении условного типа мы также можем использовать дистрибутивный синтаксис для возврата одного из нескольких типов. В качестве примера рассмотрим следующее определение функции:

```
function compareTwoValues(
```

```
    input : string | number | Date,
    compareTo : string | number | Date ) { }
```

Здесь мы определили функцию с именем `compareTwoValues`, у которой есть два параметра, `input` и `compareTo`. Оба параметра могут принимать либо тип `string`, либо `number`, либо тип `Date`. Но что, если мы хотим применить следующие правила:

- если параметр `input` имеет тип `Date`, то только параметру `compareTo` разрешается иметь тип `Date`;
- если параметр `input` имеет тип `number`, то параметру `compareTo` разрешается иметь тип `number` или `Date`;
- если параметр `input` имеет тип `string`, то параметру `compareTo` разрешается иметь тип `number`, `Date` или `string`.

При реализации этой логики пригодятся распределенные условные типы:

```
type dateOrNumberOrString<T> =
   T extends Date ? Date :
   T extends number ? Date | number :
   T extends string ? Date | number | string : never;
```

Здесь у нас есть условный тип с именем `dateOrNumberOrString`. Первое условие гласит, что если `T` имеет тип `Date`, то возвращается тип `Date`. Второе условие гласит, что если `T` имеет тип `number`, тогда возвращается тип `Date` или `number`, и последнее условие гласит, что если `T` имеет тип `string`, то возвращает тип `Date`, `number` или `string`. Если ни один из этих условных типов не найден, возвращается тип `never`. Теперь мы можем использовать этот распределенный условный тип в определении функции:

```
function compareValues<T extends string | number | Date | boolean>
   (input: T, compareTo: dateOrNumberOrString<T>) {
   // Выполняем сравнение;
}
```

Здесь мы определили функцию `CompareValues`, которая использует распределенный тип для типа `T` и позволяет типу `T` быть либо строкой, либо числом, либо датой, либо логическим типом. Параметр `input` имеет тип `T`, а параметр `compareTo` теперь использует наш распределенный условный тип `dateOrNumberOrString`. Теперь у нас есть правила времени разработки для кода, который вызывает эту функцию. Давайте проверим эту разработку, используя типы `Date`:

```
compareValues(new Date(), new Date());
compareValues(new Date(), 1);
compareValues(new Date(), "1");
```

Мы вызываем функцию `compareValues` с типом `Date` в качестве первого аргумента. Первый вызов `compareValues` допустим, так как природа второго аргумента имеет тип `Date`. Второй и третий вызовы `CompareValues`, однако, вызовут ошибки компиляции:

```
error TS2345: Argument of type '1' is not assignable to
parameter of type 'Date'.
error TS2345: Argument of type '"1"' is not assignable to
parameter of type 'Date'.
```

Здесь видно, что компилятор не допустит, чтобы аргумент `compareTo` имел тип `number` или `string`, если аргумент `input` имеет тип `Date`. Давайте теперь проверим наш распределенный условный тип с числами:

```
compareValues(1, 1);
compareValues(1, Date.now());
compareValues(1, "1");
```

Здесь типом аргумента `input` является число, и мы вызываем функцию `compareTo` с типом `number`, затем `Date`, а потом `string`. Первые два вызова являются допустимыми, согласно нашей логике, но последний вызов даст ошибку:

```
error TS2345: Argument of type '"1"' is not assignable to
parameter of type 'Date'.
```

Наконец, давайте нарушим все правила и вызовем нашу функцию `compareTo` с типом `boolean`:

```
compareValues(true, "test");
```

В данном случае тип аргумента `input` – `boolean`. Обратите внимание, что наше определение функции разрешает это, так как тип `T` может иметь тип `string`, `number`, `Date` или `boolean`. Распределенный условный тип, однако, этого не разрешает и, следовательно, выдаст ошибку:

```
error TS2345: Argument of type '"test"' is not assignable to
parameter of type 'never'.
```

Таким образом, наш распределенный условный тип проверяет аргумент `input`, который имеет тип `boolean`, не соответствует ни одному из наших условий и, следовательно, возвращает тип `never`. Наша конструкция функции сохранилась, хотя мы могли ошибиться, допустив, что аргумент `input` имеет тип `boolean`.

Выведение условных типов

При работе с условными типами мы можем извлечь тип каждого параметра, используя ключевое слово `infer`. Это лучше всего объяснить на следующих примерах:

```
type extractArrayType<T> = T extends (infer U)[] ? U : never;
let stringType : extractArrayType<["test"]> = "test";
let stringTypeNoArray : extractArrayType<"test"> = "test";
```

Здесь у нас есть условный тип с именем `extractArrayType`, который использует синтаксис обобщений и поэтому должен использоваться с типом `T`. Наш условный тип затем проверяет, является ли тип `T` массивом. Если это массив, мы используем ключевое слово `infer` для вывода типа с именем `U`; в противном случае мы возвращаем тип `never`. Обратите внимание, что имена выводимых типов (в данном случае `U`) могут использоваться только внутри условного типа. Затем мы определяем переменную с именем `stringType`, тип которой является результатом нашего условного типа `extractArrayType`. Поскольку наш тип `T` в этом случае является массивом строковых типов, тип `U` будет строковым. Это означает, что мы извлекли тип из массива, используя синтаксис (вывод `U`).

Однако последняя строка нашего фрагмента кода выдаст ошибку:

```
error TS2322: Type '"test"' is not assignable to type 'never'
```

Поэтому наш условный тип определяет, что тип `"test"` не является массивом и поэтому возвращает тип `never`.

Давайте рассмотрим несколько более сложный случай:

```
type InferredAb<T> = T extends { a: infer U, b: infer U } ? U : T;
type abInferredNumber = InferredAb< { a :number, b: number}>;
let abinf : abInferredNumber = 1;

type abInferredNumberString = InferredAb< { a :number, b: string}>;
let abinfstr : abInferredNumberString = 1;
abinfstr = "test";
```

Здесь мы определили условный тип с именем `InferredAb`, который проверяет, есть ли у нашего типа свойства `a` и `b`. Если это так, он выведет тип `U` из свойств `a` и `b`. Затем мы создаем второй тип с именем `abInferredNumber`, который использует наш условный тип с объектом, где `a` и `b` – это числа. Это означает, что `a: infer U` возвращает число и `b: infer U` также возвращает число. Следовательно, тип `U` будет типом числа. Следовательно, переменная `abinf` имеет тип `number`.

Затем мы создаем тип с именем `abInferredNumberString` и используем наш условный тип с типом, где `a` – это число, `a b` – строка. Это означает, что `a: infer U` вернет тип числа, `a b: infer U` вернет тип строки. Таким образом, результатом нашего условного типа будет объединение типов `number | string`. Это означает, что когда мы создаем переменную с именем `abinfstr` и устанавливаем для ее типа значение `abInferredNumberString`, мы можем присвоить ей либо число, либо строку, поскольку наш выводимый тип – `number | string`.

keyof

TypeScript позволяет перебирать свойства типа и извлекать имена его свойств с помощью ключевого слова `keyof`. Поэтому `keyof` вернет строковый литерал, состоящий из имен свойств. Давайте рассмотрим эту концепцию, начав с интерфейса:

```
interface IPerson {
   id: number;
   name: string;
   surname: string;
}
```

Здесь у нас есть интерфейс с именем `IPerson`, у которого есть три свойства: `id`, `name` и `surname`. Если бы нам нужно было написать функцию, для которой необходимо, чтобы ее ввод ограничивался строковыми значениями `"id"`, `"name"` и `"surname"`, мы могли бы использовать строковый литерал:

```
type PersonPropertyLiteral = "id" | "name" | "surname";

function getKeyOfUsingStringLiteral
(ppl : PersonPropertyLiteral, value : IPerson) {
   console.log(`${ppl} : ${value[ppl]}`)
}
```

Здесь мы определили строковый литерал с именем `PersonPropertyLiteral`, который ограничивает строковые значения либо `"id"`, либо `"name"`, либо `"surname"`. Затем мы используем этот строковый литерал в функции `getKeyOfUsingStringLiteral`, чтобы гарантировать, что первый параметр, `ppl`, соответствует одному из этих трех значений. Это необходимо для того, чтобы при обращении к параметру значения с помощью `value[ppl]` мы правильно использовали имя свойства, которое является частью интерфейса `IPerson`. К сожалению, если мы когда-либо изменим интерфейс `IPerson`, нам также необходимо будет не забыть изменить литерал `PersonPropertyLiteral`.

Однако ключевое слово `keyof` автоматически создает строковый литерал на основе свойств данного типа, поэтому нам не нужно создавать литералы вручную.

Рассмотрим следующую функцию:

```
function getKeyUsingKeyOf(key: keyof IPerson, value: IPerson):
void {  console.log(`${key} : ${value[key]}`); }
```

Здесь у нас есть функция `getKeyUsingKeyOf`, которая использует ключевое слово `keyof` для достижения того же результата. Тип параметра `key` теперь использует ключевое слово `keyof` для автоматической генерации строкового литерала для всех свойств интерфейса `IPerson`. Затем мы можем использовать эту функцию следующим образом:

```
let testPerson : IPerson = { id: 1, name: "test", surname: "true" };
getKeyUsingKeyOf("id", testPerson);
getKeyUsingKeyOf("name", testPerson);
getKeyUsingKeyOf("surname", testPerson);
```

Мы создаем переменную `testPerson` типа `IPerson`, которая определяет свойства `id`, `name` и `surname`. Затем мы вызываем функцию `getKeyUsingKeyOf` несколько раз, чтобы записать значение свойств `id`, `name` и `surname` в консоль. Имейте в виду, что если мы попытаемся вызвать эту функцию со свойством, которое не существует в интерфейсе `IPerson`:

```
getKeyUsingKeyOf("notaproperty", testPerson);
```

компилятор выдаст ошибку:

```
error TS2345: Argument of type '"notaproperty"' is not
assignable to parameter of type '"id" | "name" | "surname"'
```

Эта ошибка указывает на то, что у интерфейса `IPerson` нет свойства с именем `notaproperty`, поскольку в результате действия `keyof IPerson` разрешит только одно из строковых значений: `id`, `name` либо `surname`.

keyof с числом

Мы также можем использовать ключевое слово keyof при работе с числовыми свойствами объекта.

Помните, что если мы определяем объект с числовым свойством, то нам нужно как определить свойство, так и получить доступ к нему, используя синтаксис массива:

```
class ClassWithNumericProperty {
   [1] : string = "one";
}
let classWithNumeric = new ClassWithNumericProperty();
console.log(`${classWithNumeric[1]}`);
```

Здесь мы определили класс `ClassWithNumericProperty`, который определяет свойство с именем `1` типа `string` и значение `"one"`. Обратите внимание, что нам нужно определить это свойство, используя синтаксис массива, определив его как `[1]` вместо просто `1`. Затем мы создаем класс с именем `classWithNumeric` и записываем значение свойства `[1]` в консоль. Вывод этого кода:

```
one
```

Это показывает, что мы можем использовать числовые имена свойств, если используем синтаксис массива.

Учитывая это, давайте напишем код, который будет возвращать строковое значение для типа enum. Рассмотрим следующий код:

```
enum Currency {
   AUD = 36,
   PLN = 985,
   USD = 840
}

const CurrencyName = {
   [Currency.AUD]: "Australian Dollar",
   [Currency.PLN]: "Zloty"
}
```

Здесь мы определили перечисление с именем `Currency` и указали, что значения этих перечислений представляют собой соответствующие им международные коды валют. Затем мы определяем константу `CurrencyName`, которая определяет строковое значение для каждого из значений перечисления. Следовательно, мы можем найти строковое значение для значения перечисления `AUD` двумя способами:

```
console.log(`CurrencyName[Currency.AUD] =
   ${CurrencyName[Currency.AUD]}`);
console.log(`CurrencyName[36] = ${CurrencyName[36]}`);
```

Здесь мы находим строковое значение для австралийского доллара, обращаясь к соответствующему числовому свойству `CurrencyName[Currency.AUD]` или просто `CurrencyName[36]`.

Затем мы можем использовать ключевое слово `keyof` для генерации числового литерала числовых свойств объекта `CurrencyName`:

```
function getCurrencyName<T, K extends keyof T>
(key: K, map: T): T[K] { return map[key]; }
```

Здесь мы определили функцию `getCurrencyName`, которая использует синтаксис обобщений и определяет обобщенный тип с именем `T`, а также обобщенный тип с именем `K`. Ключевая часть этого фрагмента кода – тип `K extends keyof T`. Другими словами, `K` должен быть числовым литералом, полученным из числовых свойств `T`. У этой функции есть два параметра, `key` (для типа `K`) и `map` (для типа `T`). Обратите внимание, что она возвращает `T[K]`. Это означает, что она возвращает свойство с именем `K` из типа `T`. Реализация этой функции действительно проста тем, что она возвращает `map[key]`, то есть она возвращает числовое свойство с именем `[key]` из объекта с именем `map`. Интересным аспектом является то, как мы используем эту функцию:

```
let name = getCurrencyName(Currency.AUD, CurrencyName);
console.log(`name = ${name}`);
```

```
name = getCurrencyName(Currency.PLN, CurrencyName);
console.log(`name = ${name}`);
```

Здесь мы определяем переменную name, которая является результатом вызова функции getCurrencyName. Этот вызов использует два аргумента. Первый – это значение из CurrencyEnum, а второй – сам объект CurrencyName. Вывод этого кода выглядит так:

```
name = Australian Dollar
name = Zloty
```

При первом вызове функции getCurrencyName мы указываем значение перечисления Currency.AUD в качестве нашего первого аргумента, который является типом K. Затем мы указываем объект CurrencyName в качестве нашего второго аргумента, который имеет тип T. Согласно определению функции, K должен расширять keyof T. Поскольку CurrencyName[36] или CurrencyName [Currency.AUD] является свойством объекта CurrencyName, мы выполняем эти правила. Если, однако, мы попытаемся обратиться к числовому свойству, которое не существует в объекте CurrencyName:

```
name = getCurrencyName(Currency.USD, CurrencyName);
```

компилятор TypeScript выдаст следующее сообщение:

```
error TS2345: Argument of type 'Currency.USD' is not assignable
to parameter of type 'Currency.AUD | Currency.PLN'.
```

Эта ошибка указывает на то, что у объекта CurrencyName нет свойства с именем Currency.USD, поскольку ключевое слово keyof генерирует числовой литерал Currency.AUD | Currency.PLN для типа T. Таким образом, наш тип K, следовательно, не расширяет keyof T, отсюда и ошибка.

Отображаемые типы

Ключевое слово keyof и обобщенные типы позволяют заниматься довольно интересной математикой теоретических типов и отображать один тип в другой на основе простых правил. Это лучше всего проиллюстрировать на примерах:

```
interface IAbcRequired {
   a: number;
   b: string;
   c: boolean;
}

type PartialProps<T> = {
   [K in keyof T]?: T[K];
}
```

Здесь мы определили интерфейс с именем `IAbcRequired`, у которого есть три свойства с именами `a`, `b` и `c`. Затем мы определяем тип с именем `PartialProps`, который использует синтаксис обобщений, для которого требуется тип с именем `T`. Этот тип `PartialProps` определяет единственное свойство с именем `K` типа `keyof T`.

Другими словами, возьмите все свойства типа `T` и включите их в новый тип с именем `PartialProps`. Интересно, что в этом определении типа используется необязательный синтаксис (символ `?`), чтобы пометить все свойства типа `T` как необязательные.

Другими словами, мы отобразили тип `T` в новый тип и преобразовали все свойства типа `T` в необязательные свойства.

Теперь мы можем определить отображаемый тип:

```
type IPartialAbc = PartialProps<IAbcRequired>;
```

Здесь мы определили тип `IPartialAbc`, который использует наш отображаемый тип `PartialProps`, чтобы сделать все свойства интерфейса `IAbcRequired` необязательными. Затем мы можем использовать новый сопоставленный тип следующим образом:

```
let abNoCObject: IPartialAbc = { a: 1, b: "test" };
let aNoBcObject: IPartialAbc = { a: 1 };
```

Мы определили две переменные с именами `abNoCObject` и `aNoBcObject`, которые определяют свойства `a` и `b`, а затем только свойство `a` соответственно. Итак, по сути, мы использовали исходный тип `IAbcRequired` и отобразили его в другой тип, где свойства не являются обязательными.

Partial, Readonly, Record и Pick

Отображаемые типы, которые преобразуют свойства в необязательные или преобразуют свойства в readonly, считаются настолько фундаментальными, что они были включены в стандартные определения типов TypeScript. Другими словами, мы можем использовать `Partial<T>` для создания типа, в котором все свойства `T` не являются обязательными, или `ReadOnly<T>` для создания типа, где все свойства `T` – `readonly`. Чтобы создать эти отображаемые типы для нашего интерфейса `IAbcRequired`, мы можем просто написать следующее:

```
type partialAbc = Partial<IAbcRequired>;
type readonlyAbc = Readonly<IAbcRequired>;
```

Здесь тип `partalAbc` является копией типа `IAbcRequired`, но все свойства у него помечены как необязательные. Точно так же тип `readonlyAbc` является копией типа `IAbcRequired`, но со всеми свойствами, помеченными как `readonly`.

Мы также можем создать тип, который является подмножеством свойств другого типа, используя тип Pick.1 определяется следующим образом:

```
/* Выбираем набор свойств K из T; */
type Pick<T, K extends keyof T> = {
   [P in K]: T[P];
};
```

Рассмотрим следующий код:

```
type pickAb = Pick<IAbcRequired, "a" | "b">;
let pickAbObject : pickAb = { a: 1, b: "test"};
let pickAcObject : pickAb = { a : 1, c: true};
```

Здесь мы определили тип с именем pickAb, который использует общее определение Pick, чтобы создать новый тип, включающий в себя только свойства a и b IAbcRequired. Затем мы создаем объект с именем pickAbObject типа pickAb, который как таковой нуждается в свойствах a и b. Обратите внимание, что в последней строке этого фрагмента мы пытаемся создать объект с именем pickAcObject типа pickAb, но у которого есть свойства a и c. Компилятор выдаст ошибку:

```
error TS2322: Type '{a: number; c: boolean;}' is not assignable
to type 'Pick<IAbcRequired, "a" | "b">'.
```

Компилятор говорит нам, что мы не можем присвоить объект со свойствами a и c объекту, у которого есть свойства a и b.

Последний отображаемый тип, который мы рассмотрим, – это Record, который определяется так:

```
/* Создаем тип с набором свойств K типа T; */
type Record<K extends keyof any, T> = {
   [P in K]: T;
};
```

С помощью этого определения мы можем создать совершенно новый тип, указав список свойств и их тип:

```
type recordAc = Record< "a" | "c", string>;
let recordAcObject : recordAc = {a : "test", c: "test"};
let recordAcNumbers : recordAc = { a: 1, c: "test"};
```

Здесь мы определили тип с именем recordAc, который использует обобщенное определение Record для создания типа, у которого есть свойства a и c типа string. Затем мы создаем объект с именем recordAcObject, который использует этот тип и задает свойства a и c. Однако последняя строка этого кода выдаст ошибку компиляции:

```
error TS2322: Type number is not assignable to type string
```

Компилятор говорит нам, что оба свойства типа `recordAc` должны быть строками.

На этом мы завершаем наше исследование расширенных обобщений, в ходе которого мы увидели, как свойства TypeScript позволяют нам смешивать и сопоставлять типы и конструировать их из других типов, используя общий синтаксис или простой синтаксис, позволяющий нам заниматься математикой теоретических типов. Просто помните, что типы, которые мы создаем, – это просто типы, и они будут скомпилированы в результирующем JavaScript.

Асинхронное программирование

В этом разделе мы обсудим ряд асинхронных особенностей языка, в частности промисы и ключевые слова `async` и `await`.

Примеры кода, приведенные в этом разделе, были разработаны для Node версии 4 и выше, которая обеспечивает среду выполнения ECMAScript 6. Вы можете определить, какую версию Node вы используете, выполнив в командной строке:

```
node --version
```

Вы должны убедиться, что возвращаемое значение – v. 4 или выше, чтобы компилировать и запускать примеры кода, приведенные в этом разделе. Информация, возвращаемая версией Node, которая используется в этом разделе, выглядит так:

```
v. 11.9.0
```

Промисы

Промисы – это метод стандартизации асинхронной обработки в JavaScript.

Помните, что функция в JavaScript вызывается во многих случаях, но фактические результаты получаются только через определенный промежуток времени. Эти случаи обычно возникают, когда ваш код запрашивает какой-либо ресурс, например отправляет запрос на веб-сервер для получения данных JSON или читает файл с диска. Стандартный метод асинхронной обработки в JavaScript – это механизм обратного вызова.

К сожалению, при работе с большим количеством обратных вызовов наш код иногда может становиться довольно сложным и повторяющимся. Промисы предоставляют способ упростить код обратного вызова. Чтобы приступить к обсуждению промисов, давайте рассмотрим типичный код обратного вызова:

```
function delayedResponseWithCallback(callback: Function) {
  function delayedAfterTimeout() {
```

```
      console.log(`delayedAfterTimeout`);
      callback();
    }
    setTimeout(delayedAfterTimeout, 1000);
  }

  function callDelayedAndWait() {
    function afterWait() {
      console.log(`afterWait`);
    }
    console.log(`calling delayedResponseWithCallback`);
    delayedResponseWithCallback(afterWait);
    console.log(`after calling delayedResponseWithCallback`);
  }

  callDelayedAndWait();
```

Вначале идет функция `delayedResponseWithCallback`, которая принимает один аргумент типа `Function` с именем `callback`. Если мы рассмотрим последнюю строку этой функции, то увидим, что она вызывает функцию `setTimeout` с задержкой в 1000 миллисекунд, или 1 секунду. Это предназначено для имитации задержки обработки, то есть асинхронной функции в нашем коде. Функция `setTimeout` принимает функцию в качестве первого параметра, который будет вызываться после 1-секундной задержки. В этом случае мы передаем функцию с именем `delayedAfterTimeout`, которая просто записывает сообщение в консоль, а затем вызывает нашу функцию обратного вызова.

Вторая функция в этом фрагменте кода носит название `callDelayedAndWait`. Если мы посмотрим на последние три строки этой функции, то увидим, что она записывает сообщение в консоль, а затем вызывает функцию `delayedResponseWithCallback`. Она передает функцию `afterWait` в качестве функции обратного вызова. Последняя строка кода выполняет функцию `callDelayedAndWait`. Вывод этого кода выглядит так:

```
calling delayedResponseWithCallback
after calling delayedResponseWithCallback
delayedAfterTimeout
afterWait
```

Видна последовательность событий, которые происходят в нашем коде.

Когда мы выполняем функцию `callDelayedAndWait`, она устанавливает функцию обратного вызова, а затем записывает текст `calling delayedResponseWithCallback` в консоль. Далее она вызывает функцию `delayedResponseWithCallback`, а затем переходит к следующей строке, где записывает текст `after calling delayedResponseWithCallback`. После 1-секундной задержки вызывается функция `delayedAfterTimeout`, которая затем записывает

текст `delayedAfterTimeout` в консоль, и в конце вызывается функция afterWait и записывается в консоль текст `afterWait`.

Хотя такого рода функции обратного вызова являются довольно стандартными в JavaScript, это может сделать наш код трудным для чтения и понимания, особенно по мере того, как наша кодовая база будет становиться все больше и больше. С другой стороны, промисы обеспечивают свободный синтаксис для обработки асинхронных вызовов.

Синтаксис промисов

Промис – это объект, который создается путем передачи функции, которая принимает два обратных вызова.

Первый вызов используется для указания успешного ответа, а второй – для указания ошибки. Рассмотрим приведенное ниже определение функции:

```
function fnDelayedPromise (
resolve: () => void,
reject : () => void)
{
   function afterTimeout() {
      resolve();
   }
   setTimeout(afterTimeout, 2000);
}
```

Здесь мы определили функцию с именем `fnDelayedPromise`, которая принимает две функции в качестве аргументов. Эти функции называются `resolve` и `reject`, и обе они возвращают `void`.

В теле функции `fnDelayedPromise` мы снова вызываем функцию `setTimeout` (в последней строке функции), чтобы подождать 2 секунды, прежде чем вызвать функцию обратного вызова `resolve`.

Теперь мы можем использовать эту функцию для создания объекта promise:

```
function delayedResponsePromise() : Promise<void> {
   return new Promise<void>(
      fnDelayedPromise
   );
}
```

Здесь мы создали функцию с именем `delayedResponsePromise`, которая возвращает объект `new Promise<void>`. В теле функции мы просто создаем и возвращаем новый объект `promise` и используем наше более раннее определе-

ние функции с именем `fnDelayedPromise` в качестве единственного аргумента в ее конструкторе. Обратите внимание, что тип `void`, который используется для создания промиса, использует синтаксис обобщений (`new Promise<void>`), чтобы указать информацию о возвращаемом типе промиса. Мы обсудим использование общего синтаксиса `<void>` чуть позже, когда будем изучать, как возвращать значения из промисов.

Хотя этот синтаксис может показаться немного запутанным, в обычной практике эти два определения функций объединяются в один блок кода. Целью предыдущих двух фрагментов было выделить два важных понятия. Во-первых, чтобы использовать промисы, вы должны вернуть новый объект `promise`. Во-вторых, этот объект создается с функцией, которая принимает два аргумента обратного вызова.

Давайте посмотрим, как эти два шага объединяются в общей практике:

```
function delayedPromise() : Promise<void> {
   return new Promise<void>
   (
      ( resolve : () => void,
        reject: () => void
      ) => {
        function afterTimeout() {
          resolve();
        }
        setTimeout( afterTimeout, 1000);
      }
   );
}
```

Здесь у нас есть функция `delayedPromise`, которая возвращает новый объект `Promise<void>`. Первая строка этой функции создает объект `new Promise` и передает определение анонимной функции, которое принимает две функции обратного вызова, `resolve` и `reject`. Тело кода затем определяется после стрелки `=>` и заключается в соответствующие фигурные скобки `{` и `}`. Тело кода определяет функцию с именем `afterTimeout`, которая будет вызываться через 1 секунду. Обратите внимание, что функция `afterTimeout` вызывает `resolve`.

Обратите внимание, что этот фрагмент кода был тщательно отформатирован, чтобы четко показать соответствующие фигурные скобки `(` и `)` и соответствующие фигурные скобки `{` и `}`. Помните, что для использования промисов мы должны создать и вернуть объект `new Promise`, а конструктор объекта `Promise` принимает функцию (или анонимную функцию) с двумя аргументами обратного вызова.

Использование промисов

Промисы предоставляют простой синтаксис для обработки функций `resolve` и `reject`. Давайте посмотрим, как мы будем использовать промисы, определенные в нашем предыдущем фрагменте кода:

```
function callDelayedPromise() {
   console.log(`calling delayedPromise`);
   delayedPromise().then(
      () => { console.log(`delayedPromise.then()`) }
   );
}

callDelayedPromise();
```

Здесь мы определили функцию с именем `callDelayedPromise`. Эта функция записывает сообщение на консоль, а затем вызывает нашу функцию `delayedPromise`. Мы используем свободный синтаксис, чтобы присоединиться к функции промиса `then`, и определяем анонимную функцию, которая будет вызвана, когда промис будет выполнен. Вывод этого кода выглядит так:

```
calling delayedPromise
delayedPromise.then()
```

Свободный синтаксис промиса также определяет функцию `catch`, которая используется для обработки ошибок. Рассмотрим приведенное ниже определение промиса:

```
function errorPromise() : Promise<void> {
   return new Promise<void>
   (
      ( resolve: () => void,
        reject: () => void
      ) => {
         reject();
      }
   );
}
```

Здесь мы определили функцию с именем `errorPromise`, используя синтаксис промиса. Обратите внимание, что в теле функции промиса мы вызываем функцию обратного вызова `reject` вместо функции `resolve`. Функция `reject` используется для указания на ошибку. Давайте теперь используем функцию `catch`, чтобы перехватить эту ошибку:

```
function callErrorPromise() {
   console.log(`calling errorPromise`);
```

```
    errorPromise().then(
      () => { console.log(`no error.`) }
    ).catch(
      () => { console.log(`an error occurred`)}
    );
}
callErrorPromise();
```

Мы определили функцию с именем `callErrorPromise`, которая записывает сообщение в консоль и затем вызывает промис `errorPromise`. Используя свободный синтаксис, мы определили анонимную функцию, которая будет вызываться в ответе `then` (то есть в случае успеха), и также определили анонимную функцию, которая будет вызываться в ответе `catch` (то есть при ошибке). Вывод этого кода выглядит так:

```
calling errorPromise
an error occurred
```

Механизм обратного вызова в сравнении с синтаксисом промиса

Для сравнения двух методов, которые мы обсуждали, давайте посмотрим на упрощенную версию обратного вызова в сравнении с синтаксисом промиса.

Наш стандартный механизм обратного вызова выглядит следующим образом:

```
function standardCallback() {
  function afterCallbackSuccess() {
    // Выполняем этот код
  }
  function afterCallbackError() {
    // В случае ошибки
  }
  // Вызываем асинхронную функцию
  invokeAsync(afterCallbackSuccess, afterCallbackError);
}
```

А синтаксис промиса выглядит так:

```
function usingPromises() {
  delayedPromise().then(
    () => {
      // В случае успеха
    }
  ).catch (
    () => {
```

```
        // В случае ошибки
      }
    );
}
```

Как видно, использование промисов обеспечивает свободный синтаксис для обработки асинхронного программирования.

Возвращение значений из промисов

Пока что мы определили все наши promise-объекты как `Promise<void>`. `void` в данном случае означает, что наши промисы не будут возвращать никаких значений. Если мы используем `Promise<string>`, это означает, что они будут возвращать строковые значения. Давайте посмотрим, как вернуть значения из промисов:

```
function delayedPromiseWithParam() : Promise<string> {
   return new Promise<string>(
      (
         resolve: (str: string) => void,
         reject: (str:string ) => void
      ) => {
         function afterWait() {
            resolve("resolved_within_promise");
         }
         setTimeout( afterWait , 2000 );
      }
   );
}
```

Здесь у нас есть функция с именем `delayedPromiseWithParam`, которая создает и возвращает наш promise-объект, как обычно. Однако обратите внимание на то, что определение функций обратного вызова `resolve` и `reject` теперь принимает один строковый аргумент.

Этот строковый аргумент связывается с обобщенным типом, который был определен для данного промиса, то есть `Promise<string>`. Если бы мы хотели использовать числовой тип для аргументов `resolve` и `reject`, нам нужно было бы определить наш возвращаемый тип как `Promise<number>`.

Внутренняя работа анонимной функции аналогична той, что мы обсуждали ранее, за исключением того, что вызов `resolve` теперь содержит строку в качестве аргумента.

Давайте посмотрим, как можно использовать это возвращаемое значение:

```
function callPromiseWithParam() {
   console.log(`calling delayedPromiseWithParam`);
```

```
    delayedPromiseWithParam().then( (message: string) => {
      console.log(`Promise.then() returned ${message} `);
    } );
}

callPromiseWithParam();
```

Здесь мы определили функцию с именем `callPromiseWithParam`, которая записывает сообщение в консоль, а затем вызывает нашу функцию `delayedPromiseWithParam`. Далее мы используем свободный синтаксис для присоединения анонимной функции к функции промиса `then`. Обратите внимание, что теперь наша анонимная функция принимает единственный строковый параметр с именем `message`. Это соответствует `resolve: (str: sring)`. Вывод этого кода выглядит так:

```
calling delayedPromiseWithParam
Promise.then() returned resolved_within_promise
```

Как и ожидалось, наш промис вызвал функцию `resolve`, то есть `resolve ("resolved_within_promise")`, что соответствует нашему обработчику `then( (message: string) => { ... } )`.

Обратите внимание, что промисы могут возвращать только одно значение при вызове функций `resolve` и `reject`. Если вам понадобится вернуть сообщение, содержащее несколько полей, вам нужно будет использовать интерфейс:

```
interface IPromiseMessage {
  message: string;
  id: number;
}

function promiseWithInterface() : Promise<IPromiseMessage> {
  return new Promise<IPromiseMessage> (
    (
      resolve: (message: IPromiseMessage) => void,
      reject: (message: IPromiseMessage) => void
    ) => {
      resolve({message: "test", id: 1});
    }
  );
}
```

Здесь мы определили интерфейс с именем `IPromiseMessage`, который содержит сообщение типа `string` и `id` типа `number`. Наша функция `promiseWithInterface` теперь возвращает `Promise <IPromiseMessage>`, а функции обратного вызова `resolve` и `reject` используют `IPromiseMessage` в качестве типа аргумента. Наш вызов `resolve` должен теперь создать объект как со свойством `message`, так и со свойством `id`, чтобы правильно реализовать интерфейс `IPromiseMessage`.

Использование интерфейсов таким образом позволяет промисам возвращать любой тип данных.

async и await

В качестве дополнительного улучшения при работе с промисами TypeScript применяет два ключевых слова, которые работают вместе при использовании промисов. Эти два ключевых слова – async и await. Использование async и await лучше всего можно описать, рассмотрев пример кода:

```
function awaitDelayed() : Promise<void> {
   return new Promise<void> (
      (  resolve: () => void,
         reject: () => void ) =>
      {
         function afterWait() {
            console.log(`calling resolve`);
            resolve();
         }
         setTimeout(afterWait, 1000);
      }
   );
}
```

Мы начинаем с довольно стандартной функции с именем `awaitDelayed`, которая возвращает промис, похожий на те, что мы видели в предыдущих примерах. Обратите внимание, что в теле функции `afterWait` мы записываем сообщение в консоль, прежде чем вызывать функцию обратного вызова `resolve`. Давайте теперь посмотрим, как можно использовать этот промис с ключевыми словами `async` и `await`:

```
async function callAwaitDelayed() {
   console.log(`call awaitDelayed`);
   await awaitDelayed();
   console.log(`after awaitDelayed`);
}

callAwaitDelayed();
```

Вначале у нас идет функция `callAwaitDelayed`, которой предшествует ключевое слово `async`. В рамках этой функции мы записываем сообщение в консоль, а затем вызываем ранее определенную функцию `awaitDelayed`. На этот раз, однако, мы помещаем вызов перед функцией `awaitDelayed` с помощью ключевого слова `await`. Затем мы записываем другое сообщение в консоль. Вывод этого кода выглядит так:

```
call awaitDelayed
calling resolve
after awaitDelayed
```

Эти выходные данные показывают, что ключевое слово `await` фактически ожидает вызова асинхронной функции, прежде чем продолжить выполнение программы. Это обеспечивает легкий для чтения и понятный поток программной логики, автоматически приостанавливая выполнение до выполнения промиса.

awaitError

Наши promise-объекты обычно определяют как успешное условие, так и ошибочное при вызове асинхронных функций. Чтобы перехватить эти ошибки при использовании синтаксиса `async await`, мы можем использовать блок `try...catch`. Чтобы проиллюстрировать это, давайте определим промис, который возвращает ошибку, а также сообщение об ошибке:

```
function awaitError() : Promise<string> {
   return new Promise<string> (
      ( resolve: (message: string) => void,
        reject: (error: string) => void ) =>
        {
           function afterWait() {
              console.log(`calling reject`);
              reject("an error occurred");
           }
           setTimeout(afterWait, 1000);
        }
      );
}
```

Здесь у нас есть функция с именем `awaitError`, которая определяет и возвращает `Promise<string>` и использует наш стандартный синтаксис промиса с задержкой в 1 секунду. Здесь следует отметить строку внутри функции `afterWait`, где мы вызываем функцию `reject` с сообщением об ошибке. Наша соответствующая функция `async await` будет выглядеть так:

```
async function callAwaitError() {
   console.log(`call awaitError`);
   try {
      await awaitError();
   } catch (error) {
      console.log(`error returned : ${error}`);
   }
   console.log(`after awaitDelayed`);
}
callAwaitError();
```

Здесь у нас есть асинхронная функция с именем `callAwaitError`, которая записывает сообщение в консоль, а затем вызывает `await awaitError()` в блоке `try ... catch`. Еще раз обратите внимание, что программная логика будет приостановлена, когда она достигнет ключевого слова `await` для возврата асинхронной функции, прежде чем продолжить выполнение кода. В этом случае, однако, вызов `await` приведет к возникновению ошибки, которая будет перехвачена блоком `catch (error)`. В этом блоке мы записываем полученное сообщение об ошибке в консоль. Вывод этого кода выглядит так:

```
call awaitError
calling reject
error returned : an error occurred
after awaitDelayed
```

Как видно из выходных данных, выполнение программы приостанавливается для возврата асинхронной функции `await awaitError()`. При возникновении ошибки активируется блок `catch`, а аргумент error содержит сообщение об ошибке, сгенерированное в промисе.

Синтаксис промиса в сравнении с синтаксисом async await

Чтобы освежить в памяти синтаксис промисов и `async await`, давайте сравним эти два метода бок о бок. Вначале рассмотрим синтаксисы `then` и `catch`, используемые стандартными промисами:

```
function simplePromises() {
   delayedPromise().then(
      () => {
         // Выполняется при успехе
      }
   ).catch (
      () => {
         // Выполняется при ошибке
      }
   );
   // Код не ждет асинхронного вызова
   }
```

Обратите внимание, что в предыдущем коде мы используем `.then` и `.catch` для определения анонимных функций, которые будут вызываться в зависимости от того, был асинхронный вызов успешным или нет. Еще одна оговорка при использовании синтаксиса промиса – любой код вне блока `.then` или `.catch` будет выполнен немедленно и не будет ожидать завершения асинхронного вызова.

Теперь давайте рассмотрим новый синтаксис `async await`:

```
async function usingAsyncSyntax() {
   try {
      await delayedPromise();
      // Выполняется при успехе
   } catch(error) {
      // Выполняется при ошибке
   }
   // Код ждет асинхронного вызова
}
```

Данный код позволяет нам использовать очень простой синтаксис, который протекает логически. Мы знаем, что любой вызов `await` будет блокировать выполнение кода до тех пор, пока не вернется асинхронная функция, включая любой код, определенный за пределами нашего блока `try ... catch`.

Как видно из сравнения двух стилей, использование синтаксиса `async await` упрощает наш код, делает его более читабельным и менее подверженным ошибкам.

awaitMessage

Последняя тема, которую мы обсудим в свете `async await`, – способ обработки сообщений, возвращаемых в рамках наших промисов. Рассмотрим приведенное ниже определение промиса:

```
function asyncWithMessage() : Promise<string> {
   return new Promise<string> (
      (
         resolve: (message: string ) => void,
         reject: (message: string) => void
      ) => {
         function afterWait() {
            resolve("resolve_message");
         }
         setTimeout(afterWait, 1000);
      }
   );
}
```

Здесь мы определили стандартную функцию, возвращающую промис после 1-секундной задержки. На что тут стоит обратить внимание – это вызов `resolve` внутри функции `afterWait`, которая отправляет сообщение обратно в функцию обратного вызова. В этом случае она возвращает строковое значение `resolve_message`, которое соответствует нашему синтаксису `Promise<string>`. Опять же, мы можем использовать интерфейсы для возврата нескольких значений внут-

ри промиса. Наша соответствующая функция async await, которая использует этот промис, выглядит так:

```
async function awaitMessage() {
   console.log(`calling asyncWithMessage`);
   let message: string = await asyncWithMessage();
   console.log(`message returned: ${message}`);
}

awaitMessage();
```

Здесь мы определили асинхронную функцию с именем `awaitMessage`, которая записывает сообщение в консоль, а затем вызывает функцию `asyncWithMessage`. Обратите внимание, что мы получили сообщение, возвращаемое промисом, просто определяя переменную для хранения возвращаемого значения вызова `await`. Затем мы записываем полученное сообщение в консоль. Вывод этого кода выглядит так:

```
calling asyncWithMessage
message returned: resolve_message
```

Как видно из этого примера, получение и обработка сообщений, возвращаемых при использовании ключевого слова `await`, очень просты. Все, что нам нужно сделать, – это определить переменную для хранения возвращаемого результата вызова `await`, и тогда у нас будет доступ к нашему сообщению.

Резюме

В этой главе мы подробно обсудили декораторы, обобщения, расширенные обобщения и методы асинхронного программирования с использованием промисов и `async await`.

Мы увидели, как декораторы предоставляют способ внедрения кода в определения классов или их изменения. Мы также исследовали использование экспериментальной информации о метаданных при работе с декораторами и определениями классов. Затем наше обсуждение перешло к обобщениям, что это такое и как они используются. Мы работали с обобщенными интерфейсами и создавали объекты внутри обобщенных функций. После мы изучили синтаксис расширенных обобщений, включая условные и отображаемые типы. Наше последнее обсуждение касалось методов асинхронного программирования с использованием функций обратного вызова, промисов, а также ключевых слов `async` и `await`.

В следующей главе мы рассмотрим механизм, который TypeScript использует для интеграции с существующими библиотеками JavaScript, – файлы объявлений.

Глава 5

Файлы объявлений и строгие опции компилятора

Одним из наиболее привлекательных аспектов разработки на JavaScript является большое число уже опубликованных внешних библиотек JavaScript, таких как jQuery, Knockout, Underscore, Lodash или Moment. Как мы знаем, TypeScript генерирует JavaScript, поэтому мы с легкостью можем использовать библиотеки JavaScript в TypeScript. Мы уже видели, как TypeScript использует *синтаксический сахар* для улучшения нашего опыта разработки на JavaScript, предоставляя надежный механизм типизации. Однако если мы используем библиотеки JavaScript, как применить этот сахар к существующим библиотекам JavaScript или коду JavaScript? Ответ относительно прост – файлы объявлений.

Файл объявлений – это специальный тип файла, используемый компилятором TypeScript. Он помечается расширением `.d.ts`, а затем используется компилятором TypeScript в шаге компиляции. Файлы объявлений напоминают файлы заголовков, используемые в C или C++, или интерфейсы в Java. Они просто описывают синтаксис и структуру доступных функций и свойств, но не предоставляют реализацию. Следовательно, файлы объявлений фактически не генерируют никакого кода JavaScript. Они существуют просто для обеспечения совместимости TypeScript с внешними библиотеками или для заполнения пробелов в коде JavaScript, о которых TypeScript не знает. Чтобы использовать любую внешнюю библиотеку JavaScript в TypeScript, вам понадобится файл объявлений.

В компиляторе TypeScript также есть ряд опций, начиная от опций, связанных с настройкой проекта, например куда помещать скомпилированные файлы `.js`, до возможности удалять комментарии из сгенерированного кода JavaScript. В этой главе мы рассмотрим некоторые из этих опций, которые фокусируются на самом языке, а также как можно укрепить подверженный ошибкам код TypeScript с помощью этих опций.

Эта глава состоит из трех основных разделов. В первом разделе мы рассмотрим файлы объявлений, покажем их причины, а затем создадим собственный файл объявлений на основе существующего кода JavaScript. Второй раздел этой главы представляет собой краткое справочное руководство по синтаксису определения модуля. В третьем разделе мы рассмотрим ряд опций компилятора TypeScript.

Темы этой главы:

- глобальные переменные;
- использование блоков кода JavaScript в HTML;
- использование структурированных данных;
- написание файлов объявлений;
- написание интерфейсов для файлов объявлений;
- использование объединенных типов;
- слияние модулей;
- опции компилятора.

Глобальные переменные

Большинство современных веб-сайтов используют своего рода серверный движок для генерации HTML-кода своих веб-страниц. Если вы знакомы со стеком технологий Microsoft, то знаете, что ASP.NET MVC – это очень популярный серверный движок, используемый для генерации HTML-страниц на основе главных страниц, частичных страниц и представлений MVC. Если вы разработчик на Node, то можете использовать один из популярных пакетов Node, который поможет вам создавать веб-страницы с помощью таких шаблонов, как Jade, Handlebars или Embedded JavaScript (EJS).

В этих шаблонизаторах иногда может потребоваться установить свойства JavaScript на HTM-странице как результат серверной логики. В качестве примера предположим, что вы храните список контактных адресов электронной почты в своей базе данных, а затем выводите их на HTML-странице через глобальную переменную JavaScript `CONTACT_EMAIL_ARRAY`. Ваша визуализированная HTML-страница будет содержать тег `<script>`, в котором находится эта глобальная переменная и контактные адреса электронной почты. У вас может быть код JavaScript, который читает этот массив, а затем визуализирует значения в подвале сайта. Приведенный ниже пример показывает, что сгенерированный сценарий будет выглядеть так:

```
<body>
   <script type="text/javascript">
      var CONTACT_EMAIL_ARRAY = [
         "help@site.com",
         "contactus@site.com",
         "webmaster@site.com"
      ];
   </script>
</body>
```

Здесь у нас есть блок `<script>`, а внутри этого блока есть некий код JavaScript. JavaScript определяет переменную с именем `CONTACT_EMAIL_ARRAY`, которая со-

держит несколько строк. Предположим, что мы хотели написать код TypeScript, который может читать эту глобальную переменную. Рассмотрим следующий пример:

```
class GlobalLogger {
   static logGlobalsToConsole() {
      for(let email of CONTACT_EMAIL_ARRAY) {
         console.log(`found contact: ${email}`);
      }
   }
}

window.onload = () => {
   GlobalLogger.logGlobalsToConsole();
}
```

В этом коде создается класс с именем `GlobalLogger`, который содержит статическую функцию `logGlobalsToConsole`. Эта функция просто перебирает глобальную переменную `CONTACT_EMAIL_ARRAY` и записывает элементы в массиве в консоль.

Если мы скомпилируем данный код, то получим ошибку:

```
error TS2304: Cannot find name 'CONTACT_EMAIL_ARRAY'
```

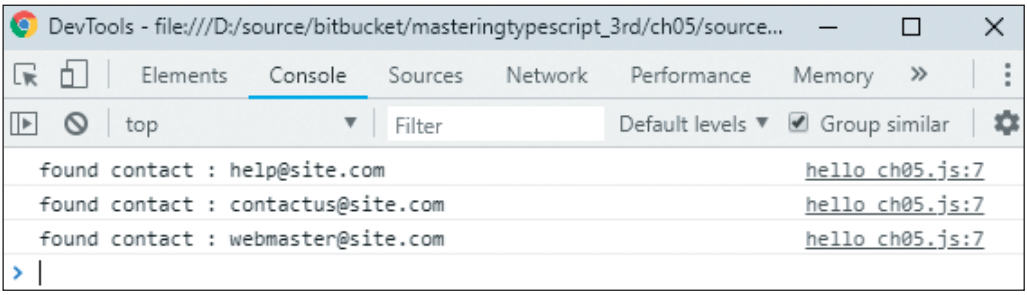
Эта ошибка указывает на то, что компилятор TypeScript ничего не знает о переменной с именем `CONTACT_EMAIL_ARRAY`. Он даже не знает, что это массив. Поскольку этот фрагмент JavaScript находится за пределами любого кода TypeScript, нам нужно будет рассматривать его так же, как и внешний код JavaScript.

Чтобы решить проблему компиляции и сделать эту переменную `CONTACT_EMAIL_ARRAY` видимой для TypeScript, нам понадобится файл объявлений. Давайте создадим файл с именем `globals.d.ts` и включим в него следующее объявление TypeScript:

```
declare var CONTACT_EMAIL_ARRAY: string [];
```

Первое, на что следует обратить внимание, – это то, что мы используем новое ключевое слово TypeScript – `declare`. Ключевое слово declare сообщает компилятору TypeScript, что мы хотим определить тип чего-либо, но реализация этого объекта (или переменной, или функции) будет завершена во время выполнения. Мы объявили переменную с именем `CONTACT_EMAIL_ARRAY` типа `string []`. Ключевое слово `declare` делает за нас две вещи: оно позволяет использовать переменную `CONTACT_EMAIL_ARRAY` в коде TypeScript, а также строго типизирует ее в виде массива строк.

Когда файл `globals.d.ts` готов, наш код компилируется правильно. Если мы теперь запустим его в браузере, вывод консоли нашего журнала браузера будет выглядеть так:



Таким образом, используя файл объявлений `globals.d.ts`, мы смогли описать структуру внешней переменной JavaScript для компилятора TypeScript. Эта переменная JavaScript определяется вне любого нашего кода TypeScript, но мы все еще можем работать с определением этой переменной в TypeScript.

Вот для чего используются файлы объявлений. В основном мы говорим компилятору TypeScript использовать определения, найденные в файле объявлений в шаге компиляции, и что сами фактические переменные будут доступны только во время выполнения.

Файлы определений также предоставляют технологию автодополнения кода IntelliSense для нашей интегрированной среды разработки при работе с внешними библиотеками JavaScript и кодом.

Использование блоков кода JavaScript в HTML

То, что мы только что видели, является примером тесной связи между сгенерированным HTML-контентом (который содержит код JavaScript в блоках сценариев) на вашей веб-странице и фактическим использованием JavaScript. Однако вы можете поспорить, говоря, что это недостаток дизайна. Если веб-странице требуется массив контактных адресов электронной почты, то JavaScript-приложение должно просто отправить AJAX-запрос на сервер для получения той же информации в формате JSON. Хотя это очень веский аргумент, в некоторых случаях включение контента в визуализированный HTML-код на самом деле выполняется быстрее.

Было время, когда интернет, казалось, был способен отправлять и получать огромное количество информации в мгновение ока. Пропускная способность и скорость интернета росли в геометрической прогрессии, а настольные компьютеры получали более крупные объемы оперативной памяти и более быстрые процессоры. Будучи разработчиками на этом этапе интернет-хайвея, мы перестали думать о том, сколько оперативной памяти было у обычного пользователя в его компьютере. Мы также перестали думать о том, сколько данных мы посылаем по

сети, потому что скорость интернета была очень быстрой и скорость обработки браузеров, казалось, была безгранична.

А потом появился мобильный телефон, и мы почувствовали, что вернулись в 90-е годы, с невероятно медленным интернет-соединением, крошечными разрешениями экрана, ограниченной вычислительной мощностью и очень небольшим объемом оперативной памяти (и популярными аркадами, такими как *Elevator Action* (`https://archive.org/details/Elevator_Action_1985_Sega_Taito_JP_en`)). Суть этой истории в том, что, будучи современными веб-разработчиками, мы все еще должны помнить о браузерах, которые работают на мобильных телефонах. Эти браузеры иногда работают на очень ограниченных интернет-соединениях, что означает, что мы должны тщательно измерять размер наших библиотек JavaScript, данных в формате JSON- и HTML-страниц, чтобы обеспечить быстрое и удобное использование наших приложений даже в мобильных браузерах. Чтобы веб-сайт был действительно глобальным, он должен работать в тех регионах мира, где интернет-соединение медленное или может прерываться, и при минимально возможных технических характеристиках оборудования. Постарайтесь запомнить, что пользователь вашего сайта может находиться в небольшом городке в отдаленной австралийской глубинке, где ему нужно поместить свой интернет-модем в спутниковую антенну и направить ее в небо, чтобы получить надежное спутниковое интернет-соединение.

Метод включения переменных JavaScript или статических JSON-данных меньшего размера в визуализированную HTML-страницу часто дает нам самый быстрый способ визуализации экрана в более старом браузере или в современном мире, на мобильном телефоне. Многие популярные сайты используют этот метод для быстрой визуализации общей структуры страницы (верхний колонтитул, боковые панели и нижние колонтитулы), прежде чем будет доставлено основное содержимое с помощью асинхронных JSON-запросов. Этот метод хорошо работает, поскольку позволяет быстрее визуализировать страницу и дает пользователю более быструю визуальную обратную связь.

Структурированные данные

Давайте расширим этот простой массив с контактными адресами электронной почты чуть более актуальными данными. Для каждого из этих адресов предположим, что мы также хотим включить некоторый текст, который будет визуализирован в подвале нашей страницы, наряду с адресами электронной почты. Рассмотрим приведенный ниже код:

```
<script type="text/javascript">
   var CONTACT_DATA = [
      { DisplayText: 'Help',
      Email: 'help@site.com' } ,
      { DisplayText: 'Contact Us',
```

```
      Email: 'contactus@site.com' },
      { DisplayText: 'Webmaster',
      Email: 'webmaster@site.com' }
   ];
</script>
```

Здесь мы определили глобальную переменную с именем CONTACT_DATA, которая представляет собой массив объектов.

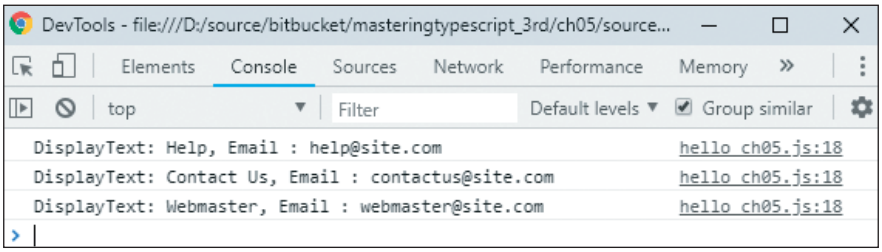
У каждого объекта есть свойство DisplayText и свойство Email. Если мы хотим использовать этот массив в своем коде TypeScript, нам нужно будет включить определение этой переменной в наш файл объявления globals.d.ts:

```
interface IContactData {
   DisplayText: string;
   Email: string; }

declare var CONTACT_DATA: IContactData[];
```

Вначале у нас идет определение интерфейса с именем IContactData для представления свойств отдельного элемента в массиве CONTACT_DATA. У каждого элемента есть свойство DisplayText типа string и свойство Email, также типа string. Поэтому наш интерфейс IContactData соответствует исходным свойствам объекта одного элемента в массиве CONTACT_DATA. Затем мы объявляем переменную с именем CONTACT_DATA и устанавливаем ее тип как массив интерфейсов IContactData, что позволяет нам работать с переменной CONTACT_DATA в TypeScript. Давайте теперь создадим класс для обработки этих данных:

```
class ContactLogger {
   static logContactData() {
      for (let contact of CONTACT_DATA) {
         console.log(`DisplayText: ${contact.DisplayText},
            Email : ${contact.Email}`);
         }
      }
}

window.onload = () => {
   ContactLogger.logContactData();
}
```

Здесь у класса ContactLogger есть единственный статический метод с именем logContactData. В этом методе мы перебираем все элементы в массиве CONTACT_DATA. Поскольку мы используем синтаксис for...of, переменная contact будет строго типизированным типом IContactData, и, следовательно, у нее будет два свойства: DisplayText и Email. Мы просто запишем эти значения в консоль. Вывод этого кода будет таким:



Пишем свой файл объявлений

В любой группе разработчиков наступает время, когда вам нужно будет либо исправить ошибку, либо усовершенствовать код, уже написанный на JavaScript. Если вы находитесь в подобной ситуации, то можете попробовать написать новые фрагменты кода на TypeScript и интегрировать их с существующим телом JavaScript. Однако для этого вам понадобится написать собственные файлы объявлений для любого существующего кода JavaScript, который вам нужно использовать повторно. Это может показаться сложной и трудоемкой задачей, но когда вы сталкиваетесь с такой ситуацией, просто не забывайте делать небольшие шаги и определять небольшие участки кода за раз. Вы будете удивлены, насколько все просто.

В этом разделе предположим, что вам нужно интегрировать существующий вспомогательный класс, который повторно используется во многих проектах, который хорошо протестирован и является стандартом группы разработчиков. Этот класс был реализован как замыкание JavaScript:

```
ErrorHelper = (function() {
   return {
      containsErrors: function (response) {
         if (!response || !response.responseText)
            return false;
         var errorValue = response.responseText;
         if (String(errorValue.failure) == "true"
            || Boolean(errorValue.failure)) {
            return true;
         }
         return false;
      },
      trace: function (msg) {
         var traceMessage = msg;
         if (msg.responseText) {
            traceMessage = msg.responseText.errorMessage;
         }
```

```
        console.log("[" + new Date().toLocaleDateString()
          + "] " + traceMessage);
      }
    }
})();
```

Этот фрагмент кода JavaScript определяет объект JavaScript с именем `ErrorHelper`, который содержит два метода. Метод `containsErrors` принимает объект с именем `response` в качестве аргумента. Этот объект затем проверяется на предмет наличия ошибок. У объекта нет ошибки, если выполнены следующие условия:

- аргумент `response` не определен;
- `response.responseText` не определен.

Однако возвращается условие ошибки, если выполнены следующие условия:

- свойство `response.responseText.failure` имеет строковое значение `"true"`;
- свойство `response.responseText.failure` имеет логическое значение `true`.

В замыкании `ErrorHelper` также есть функция `trace`, которую можно вызывать со строкой, или объект `response`, аналогичный тому, что ожидает функция `containsErrors`.

К сожалению, в функции `ErrorHelper` отсутствует ключевая часть документации. Какова структура объекта, передаваемого в эти два метода, и какими свойствами он обладает? Без какой-либо формы документации мы вынуждены выполнять обратную разработку кода, чтобы определить, как выглядит структура объекта `response`. Если мы сможем найти примеры использования класса `ErrorHelper`, это может помочь нам угадать данную структуру.

В качестве примера использования `ErrorHelper` рассмотрим следующий код:

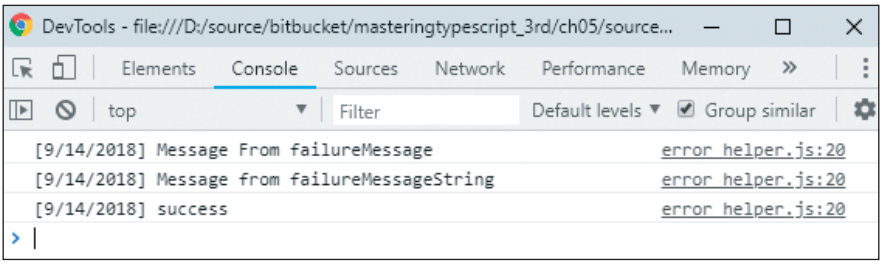
```
var failureMessage = {
  responseText : {
    "failure": true,
    "errorMessage": "Message From failureMessage"
  }
}

var failureMessageString = {
  responseText: {
    "failure": "true",
    "errorMessage": "Message from failureMessageString"
  }
}
```

```
var successMessage = {
   responseText: {
      "failure": false
   }
}

if (ErrorHelper.containsErrors(failureMessage))
   ErrorHelper.trace(failureMessage);
if (ErrorHelper.containsErrors(failureMessageString))
   ErrorHelper.trace(failureMessageString);
if (!ErrorHelper.containsErrors(successMessage))
   ErrorHelper.trace("success");
```

Вначале у нас идет переменная `failMessage`, у которой одно свойство, `reponseText`. У этого свойства, в свою очередь, есть два дочерних свойства – `failure` и `errorMessage`. Наша следующая переменная `faultMessageString` имеет такую же структуру, но определяет свойство `responseText.failure` как свойство типа `string`, а не типа `boolean`. Наконец, наш объект `successMessage` определяет, что свойство `responseText.failure` должно иметь значение `false`, но у него нет свойства `errorMessage`.

В формате JSON имена свойств должны заключаться в кавычки, тогда как в формате объектов JavaScript они являются необязательными. Следовательно, структура `{"fail": true}` синтаксически эквивалентна структуре `{fail: true}`.

Последние несколько строк предыдущего блока кода показывают, как используется замыкание `ErrorHelper`. Все, что нам нужно сделать, – это вызвать метод `ErrorHelper.containsErrors` с нашей переменной и, если результат равен `true`, записать сообщение в консоль через функцию `ErrorHelper.trace`. Результат будет таким:



Ключевое слово module

Чтобы проверить замыкание `ErrorHelper` с использованием TypeScript, нам понадобится HTML-страница, которая включает в себя как файл `error_helper.js`, так и файл JavaScript, сгенерированный TypeScript.

Предположим, что наш файл TypeScript называется `ErrorHelperTypeScript.ts`. Тогда наша HTML-страница будет выглядеть так:

```
<!DOCTYPE html>
<html>
   <head>
      <title></title>
      <script src="error_helper.js"></script>
      <script src="ErrorHelperTypeScript.js"></script>
   </head>
   <body>
   </body>
</html>
```

Этот HTML-код очень прост и включает в себя как существующий файл JavaScript `error_helper.js`, так и файл `ErrorHelperTypeScript.js`, сгенерированный TypeScript.

Давайте используем `ErrorHelper` в файле `ErrorHelperTypeScript.ts`:

```
window.onload = () => {
   var failureMessage = {
      responseText : {
         "failure" : true,
         "errorMessage" : "Error Message from Typescript"
      }
   }
if (ErrorHelper.containsErrors(failureMessage))
   ErrorHelper.trace(failureMessage);
}
```

Это урезанная версия нашего исходного примера JavaScript. Сначала мы создаем объект `faultMessage` с правильными свойствами, а затем просто вызываем методы `ErrorHelper.containsErrors` и `ErrorHelper.trace`. Если бы на этом этапе мы скомпилировали наш файл TypeScript, то получили бы следующую ошибку:

```
error TS2304: Cannot find name 'ErrorHelper'.
```

Эта ошибка указывает на то, что не существует допустимого типа TypeScript с именем `ErrorHelper`, хотя у нас есть полный исходник `ErrorHelper` в нашем файле JavaScript. TypeScript по умолчанию просмотрит все файлы TypeScript в нашем проекте, чтобы найти определения классов, но он не будет анализировать файлы JavaScript. Нам понадобится новый файл определения TypeScript, чтобы правильно скомпилировать этот код.

Данный файл определения вообще не включен в HTML-файл; он используется только компилятором TypeScript и не генерирует никакого кода JavaScript.

Без набора полезной документации по нашему классу `ErrorHelper` нам нужно будет выполнить обратное проектирование определения TypeScript, просто прочитав исходный код. Эта ситуация, очевидно, не идеальна, и делать этого не рекомендуется, но на данном этапе это все, что мы можем сделать. В таких случаях лучше всего просто взглянуть на примеры использования и идти дальше.

Глядя на использование замыкания `ErrorHelper` в JavaScript, мы видим, что есть два ключевых элемента, которые должны быть включены в наш файл объявлений. Первый – это набор определений функций для `containsErrors` и `trace`. Второй – это набор интерфейсов для описания структуры объекта `response`, на который опирается замыкание `ErrorHelper`.

Давайте начнем с определений функций и создадим новый файл TypeScript, который мы назовем `ErrorHelper.d.ts` и который будет содержать следующий код:

```
declare module ErrorHelper {
   function containsErrors(response: any): boolean;
   function trace(message: any): void;
}
```

Этот файл объявления начинается с ключевого слова `declare`, которое мы видели ранее, а затем используется новое ключевое слово TypeScript – `module`. За ним должно следовать имя модуля, в данном случае это `ErrorHelper`. Имя модуля должно соответствовать имени замыкания из исходного кода JavaScript, который мы описываем. Когда мы использовали `ErrorHelper`, то всегда ставили перед функциями `containsErrors` и `trace` имя замыкания `ErrorHelper`. Этот модуль также известен как пространство имен. Если бы у нас был еще один класс с именем `AjaxHelper`, который также включал бы в себя функцию `containsErrors`, мы могли бы различать функции `AjaxHelper.containsErrors` и `ErrorHelper.containsErrors`, используя эти пространства имен или имена модулей.

Вторая строка объявления нашего модуля указывает на то, что мы определяем функцию с именем `containsErrors`, которая принимает один параметр и возвращает логический тип данных. Третья строка указывает на то, что мы определяем еще одну функцию с именем `trace`, которая также принимает один параметр, но не возвращает значение. После того как мы разобрались с определением, наш образец кода TypeScript будет скомпилирован правильно.

Интерфейсы

Хотя мы правильно определили две функции, которые доступны пользователям замыкания `ErrorHelper`, мы упускаем второй фрагмент информации об этих функциях – структуру аргумента `response`. Мы не строго типизируем аргументы для функций `containsErrors` и `trace`. На этом этапе наш код TypeScript может передавать что-либо в эти две функции, потому что у него нет определения аргу-

ментов `response` или `message`. Однако мы знаем, что обе эти функции запрашивают наши параметры для определенной структуры. Если мы передадим объект, который не соответствует этой структуре, то наш код JavaScript выдаст ошибки во время выполнения.

Чтобы решить эту проблему и сделать наш код более стабильным, давайте определим интерфейс для этих параметров:

```
interface IResponse {
   responseText: IFailureMessage;
}

interface IFailureMessage {
   failure: boolean;
   errorMessage: string;
}
```

Вначале у нас идет интерфейс с именем `IResponse`, у которого есть одно свойство `responseText` – то же имя, что и у исходного объекта JavaScript. Свойство `responseText` строго типизировано и имеет тип `IFailureMessage`. Интерфейс `IFailureMessage` строго типизирован, и у него два свойства – `failure` типа `boolean` и `errorMessage` типа `string`. Эти интерфейсы правильно описывают надлежащую структуру аргумента `response` для функций `containsErrors` и `trace`. Теперь мы можем изменить наше оригинальное объявление этих функций:

```
declare module ErrorHelper {
   function containsErrors(response: IResponse): boolean;
   function trace(message: IResponse): void;
}
```

Теперь в определении функций `containsErrors` и `trace` строго типизируется аргумент `response` типа `IResponse`, который мы определили ранее. Эта модификация файла определения вынудит при любом последующем использовании функций `containsErrors` или `trace` отправлять действительный аргумент, соответствующий структуре `IResponse`. Давайте напишем намеренно неправильный код TypeScript и посмотрим, что произойдет:

```
var anotherFailure: IResponse = {
   responseText: {
      success: true
   }
}
```

Вначале мы создаем переменную с именем `anotherFailure` и указываем, что она будет иметь тип `IResponse`. К сожалению, в интерфейсе `IResponse` нет свойства с именем `success`, поэтому появится следующая ошибка TypeScript:

```
λ tsc
ErrorHelperTypeScript.ts(16,13): error TS2322: Type '{ responseText:
{ success: boolean; }; }' is not assignable to type 'IResponse'.
  Types of property 'responseText' are incompatible.
    Type '{ success: boolean; }' is not assignable to type 'IFailureM
essage'.
      Object literal may only specify known properties, and 'success'
 does not exist in type 'IFailureMessage'.
```

Как видно из этого довольно многословного, но информативного сообщения, все ошибки вызывает структура переменной `anotherFailure`. Несмотря на то что мы правильно обратились к свойству `responseText`, свойство `responseText` – это строго типизированный тип `IFailureMessage`, которому необходимы и свойство `failure`, и свойство `errorMessage`; отсюда и ошибка.

Создав файл объявления с сильной типизацией для существующего класса `ErrorHelper`, мы можем гарантировать, что дальнейшее использование существующего JavaScript-замыкания `ErrorHelper` в TypeScript не приведет к ошибкам во время выполнения.

Объединенные типы

Мы еще не совсем закончили с файлом объявления для класса `ErrorHelper`. Если мы посмотрим на исходное использование класса `ErrorHelper` в JavaScript, то заметим, что функция `containsErrors` также позволяет свойству `error` в `responseText` быть строкой:

```
var failureMessage = {
   responseText: {
      "failure": "true",
      "errorMessage": "Error Message from Typescript"
   }
}
```

Если мы скомпилируем этот код сейчас, то получим ошибку:

```
λ tsc
ErrorHelperTypeScript.ts(11,36): error TS2345: Argument of type '{ re
sponseText: { "failure": string; "errorMessage": string; }; }' is not
 assignable to parameter of type 'IResponse'.
  Types of property 'responseText' are incompatible.
    Type '{ "failure": string; "errorMessage": string; }' is not assi
gnable to type 'IFailureMessage'.
      Types of property 'failure' are incompatible.
        Type 'string' is not assignable to type 'boolean'.
ErrorHelperTypeScript.ts(12,27): error TS2345: Argument of type '{ re
sponseText: { "failure": string; "errorMessage": string; }; }' is not
 assignable to parameter of type 'IResponse'.
```

В предыдущем определении переменной faultMessageString тип свойства "failure" – "true", т. е. это string, а не true типа boolean. Чтобы учесть этот вариант в исходном интерфейсе IFailureMessage, нам нужно изменить наш файл объявлений. Самый простой способ разрешить оба типа – использовать объединение типов:

```
interface IFailureMessage {
   failure: boolean | string;
   errorMessage: string;
}
```

Здесь мы просто обновили свойство failure в интерфейсе IFailureMessage, чтобы разрешить как логические, так и строковые типы. Наш код теперь скомпилируется правильно.

Это завершает наш файл определения для класса ErrorHelper.

Слияние модулей

Как мы теперь знаем, компилятор TypeScript автоматически найдет все файлы .d.ts в нашем проекте, чтобы подобрать файлы объявлений. Если эти файлы объявлений содержат одно и то же имя модуля, компилятор TypeScript объединит два файла и будет использовать объединенную версию объявлений модуля.

Представьте, что у нас есть файл с именем MergedModule1.d.ts, содержащий следующее определение:

```
declare module MergedModule {
   function functionA(): void;
}
```

И у нас также есть второй файл с именем MergedModule2.d.ts, который содержит такое определение:

```
declare module MergedModule {
   function functionB(): void;
}
```

Теперь компилятор TypeScript объединит эти два модуля, как если бы они были одним определением:

```
declare module MergedModule {
   function functionA(): void;
   function functionB(): void;
}
```

Это позволит `functionA` и `functionB` быть действительными функциями одного и того же пространства имен `MergedModule` и разрешить следующее использование:

```
MergedModule.functionA();
MergedModule.functionB();
```

Модули также могут объединяться с интерфейсами и перечислениями. Однако мы не можем использовать этот вид синтаксиса слияния с классами.

Справочник синтаксиса объявлений

При создании файлов объявлений и использовании ключевого слова `module` существует ряд правил, которые можно использовать для смешивания и сопоставления определений. Как программист на TypeScript вы, как правило, время от времени будете писать только определения модулей, а иногда вам будет необходимо добавить новое определение в существующий файл объявлений.

Поэтому этот раздел представляет собой краткое справочное руководство по синтаксису файлов объявлений, или шпаргалку. Каждый раздел содержит описание правила определения модуля, фрагмент синтаксиса JavaScript, а затем эквивалентный синтаксис файла объявлений TypeScript.

Чтобы использовать этот справочный раздел, просто сопоставьте код JavaScript, который вы пытаетесь эмулировать, из раздела синтаксиса JavaScript, а затем напишите файл объявления, используя эквивалентный синтаксис определений. Мы начнем с синтаксиса переопределения функций в качестве примера.

Переопределение функций

Файлы объявлений могут включать в себя несколько определений для одной и той же функции. Если одну и ту же функцию JavaScript можно вызывать с разными типами, вам необходимо будет объявить переопределение функции для каждого варианта функции.

Синтаксис JavaScript

```
trace("trace a string");
trace(true);
trace(1);
trace({ id: 1, name: "test" });
```

Синтаксис файла объявлений

```
declare function trace(arg: string | number | boolean );
declare function trace(arg: { id: number; name: string });
```

Каждое определение функции должно иметь уникальную сигнатуру функции.

Вложенные пространства имен

Определения модулей могут содержать определения вложенных модулей, которые затем преобразуются во вложенные пространства имен. Если в вашем JavaScript используются пространства имен, вам необходимо определить объявления вложенных модулей, чтобы они соответствовали пространствам имен JavaScript.

Синтаксис JavaScript

```
FirstNamespace.SecondNamespace.ThirdNamespace.log("test");
```

Синтаксис файла объявлений

```
declare module FirstNamespace {
   module SecondNamespace {
      module ThirdNamespace {
         function log(msg: string);
      }
   }
}
```

Классы

Определения классов разрешены в определениях модулей. Если ваш JavaScript использует классы или оператор new, то в вашем файле объявлений должны быть определены новые классы.

Синтаксис JavaScript

```
var myClass = new MyClass();
```

Синтаксис файла объявлений

```
declare class MyClass { }
```

Пространства имен классов

Определения классов разрешены в определениях вложенных модулей. Если ваши классы JavaScript имеют предшествующее пространство имен, вам нужно сначала объявить вложенные модули, чтобы они соответствовали пространствам имен, а затем вы можете объявить классы в правильном пространстве имен.

Синтаксис JavaScript

```
var myNestedClass = new OuterName.InnerName.NestedClass();
```

Синтаксис файла объявлений

```
declare module OuterName {
   module InnerName {
      class NestedClass {}
   }
}
```

Перегрузки конструктора класса

Определения классов могут содержать перегрузки конструктора. Если ваши классы JavaScript могут быть построены с использованием разных типов или с несколькими параметрами, вам нужно будет перечислить каждый из этих вариантов в файле объявлений как перегрузки конструктора.

Синтаксис JavaScript

```
var myClass = new MyClass();
var myClass2 = new MyClass(1, "test");
```

Синтаксис файла объявлений

```
declare class MyClass {
   constructor(id: number, name: string);
   constructor();
}
```

Свойства класса

Классы могут содержать свойства. Вам нужно будет перечислить каждое свойство вашего класса в объявлении класса.

Синтаксис JavaScript

```
var classWithProperty = new ClassWithProperty();
classWithProperty.id = 1;
```

Синтаксис файла объявлений

```
declare class ClassWithProperty {
   id: number;
}
```

Функции класса

Классы могут содержать функции. Вам нужно будет перечислить каждую функцию вашего класса JavaScript в объявлении класса, чтобы компилятор TypeScript мог принимать вызовы этих функций.

Синтаксис JavaScript

```
var classWithFunction = new ClassWithFunction();
classWithFunction.functionToRun();
```

Синтаксис файла объявлений

```
declare class ClassWithFunction {
   functionToRun(): void;
}
```

Функции или свойства, которые считаются закрытыми, не нужно раскрывать через файл объявлений, их можно просто опустить.

Статические свойства и функции

Методы и свойства класса могут быть статическими. Если ваши функции или свойства JavaScript могут быть вызваны без использования экземпляра объекта для работы, эти свойства или функции должны быть помечены как статические.

Синтаксис JavaScript

```
StaticClass.staticId = 1;
StaticClass.staticFunction();
```

Синтаксис файла объявлений

```
declare class StaticClass {
   static staticId: number;
   static staticFunction();
}
```

Глобальные функции

Функции, которые не имеют префикса пространства имен, могут быть объявлены в глобальном пространстве имен. Если ваш JavaScript определяет глобальные функции, они должны быть объявлены без пространства имен.

Синтаксис JavaScript

```
globalLogError("test");
```

Синтаксис файла объявлений

```
declare function globalLogError(msg: string);
```

Сигнатуры функций

Функция может использовать сигнатуру в качестве параметра. Функции JavaScript, которые используют функции обратного вызова или анонимные функции, должны быть объявлены с правильной сигнатурой функции.

Синтаксис JavaScript

```
describe("test", function () {
});
```

Синтаксис файла объявлений

```
declare function describe(name: string, functionDef: () =>
void);
```

Необязательные свойства

Классы или функции могут содержать необязательные свойства. Если параметры объекта JavaScript не являются обязательными, они должны быть помечены как необязательные свойства в объявлении.

Синтаксис JavaScript

```
var classWithOpt = new ClassWithOptionals();
var classWithOpt1 = new ClassWithOptionals(
   {id: 1});
var classWithOpt2 = new ClassWithOptionals(
   {name: 'test'});
var classWithOpt3 = new ClassWithOptionals(
   {id: 1, name: 'test'});
```

Синтаксис файла объявлений

```
interface IOptionalProperties {
   id?: number;
   name?: string;
}

declare class ClassWithOptionals {
   constructor(options?: IOptionalProperties);
}
```

Слияние функций и модулей

Определение функции с определенным именем может быть объединено с определением модуля с тем же именем. Это означает, что если ваша функция JavaScript может быть вызвана с параметрами и у нее также есть свойства, то вам нужно будет объединить функцию с модулем.

Синтаксис JavaScript

```
fnWithProperty(1);
fnWithProperty.name = "name";
```

Синтаксис файла объявлений

```
declare function fnWithProperty(id: number);
declare module fnWithProperty {
   var name: string;
}
```

Строгие опции компилятора

Компилятор TypeScript позволяет настраивать ряд опций компилятора.

Их можно сгруппировать в пять основных групп:

- базовые опции;
- строгие проверки типов;
- опции разрешения модулей;
- опции карты кода;
- экспериментальные опции.

Мы уже познакомились с некоторыми из этих опций, в том числе с использованием карт кода и экспериментальных опций, связанных с декораторами. В этом разделе, однако, мы сосредоточимся на строгих опциях проверки типов, которые влияют на то, как мы пишем наш код TypeScript. Эти параметры предоставляют нам проверку во время компиляции, чтобы убедиться, что мы не нарушаем некоторые основные правила TypeScript, и помогают нам писать более защищенный код. В частности, мы сосредоточимся на следующих опциях:

- `noImplicitAny`;
- `strictNullChecks`;
- `strictPropertyInitialization`;
- `noImplicitThis`;
- `noUnusedLocals`;
- `noUnusedParameters`;
- `noImplicitReturns`;
- `noFallthroughCasesInSwitch`;
- `strictBindCallApply`.

Каждый из этих параметров компилятора можно найти в файле `tsconfig.json`, который генерируется, когда мы инициализируем проект TypeScript с помощью команды `tsc --init`. Мы можем легко включить или отключить эти параметры, просто комментируя или отменяя комментирование соответствующей строки в файле `tsconfig.json`.

Обратите внимание, что существует один параметр компилятора под названием `strict`, который включает все опции компиляции, которые мы будем обсуждать. Рекомендуется оставить для этого параметра значение `true`, чтобы использовать все строгие параметры компиляции, которые TypeScript вносит в таблицу. Если вы переносите старый код для использования этих новых более строгих опций, то в этом разделе будет объяснено влияние каждой из этих опций на наш код.

noImplicitAny

Опция компилятора `noImplicitAny` используется для проверки того, не были ли объявления функций или выражения строго типизированы, и поэтому (по умолчанию) будет возвращать тип `any`.

Это можно объяснить с помощью простого примера:

```
declare function testImplicitAny();
function testNoType(value) {
}
```

Вначале идет объявление функции с именем `testImplicitAny`. Затем мы создаем функцию `testNoType` с единственным параметром `value`. Если мы скомпилируем этот код с опцией `noImplicitAny`, установленной в значение true, компилятор выдаст следующие ошибки:

```
error TS7010: 'testImplicitAny', which lacks return-type
annotation, implicitly has an 'any' return type.
error TS7006: Parameter 'value' implicitly has an 'any' type.
```

Первая ошибка говорит нам, что мы объявили функцию, но не указали, какой тип эта функция будет возвращать. Даже если функция ничего не возвращает, нам нужно указать `void` в качестве возвращаемого типа.

Вторая ошибка говорит нам, что для параметра `value` не указан тип. Мы можем легко исправить эти ошибки, вставив правильные типы:

```
declare function testImplicitAny(): void;
function testNoType(value: string) { }
```

Здесь мы указали, что тип возвращаемого значения функции `testImplicitAny` должен иметь тип `void`, и указали, что тип параметра `value` для функции `testNoType` должен иметь тип `string`.

strictNullChecks

Опция компилятора `strictNullChecks` используется для поиска в нашем коде экземпляров переменных, в которых значение переменной может быть нулевым или неопределенным во время использования. Это означает, что когда переменная фактически используется, если она не была правильно инициализирована, компилятор выдаст сообщение об ошибке. Рассмотрим следующий код:

```
let a : number;
let b = a;
```

Здесь мы определили переменную с именем `a` типа `number`. Затем мы определили переменную с именем `b` и присвоили ей значение `a`. Если опция `strictNullChecks` включена, этот код выдаст ошибку:

```
error TS2454: Variable 'a' is used before being assigned.
```

Эта ошибка говорит о том, что мы пытаемся использовать значение `a` до того, как ему будет присвоено значение любого рода. Другими словами, в то время когда мы используем переменную `a`, она все еще может быть неопределенной.

Есть два способа удалить это сообщение. Во-первых, мы можем убедиться, что переменная была присвоена до ее использования:

```
let c : number = 2;
let d = c;
```

Здесь мы определили переменную с именем с типа `number` и присвоили ей значение 2.

Затем мы создаем переменную с именем `d` и присваиваем ей значение `c`. Это не вызовет ошибку, поскольку переменной `c` было присвоено значение до ее использования.

Другой способ исправить сообщение об ошибке – сообщить компилятору, что в этом случае нам хорошо известно, что переменная может быть неопределенной:

```
let e : number | undefined;
let f = e;
```

Здесь мы определяем переменную с именем `e` типа `number` или `undefined`. Затем мы создаем переменную с именем `f` и присваиваем ей значение `e`. Обратите внимание на тонкую разницу в типах переменной `e`. Говоря компилятору, что переменная `e` может быть числом или неопределенной, мы явно заявляем, что знаем, что `e` может быть неопределенной, и позаботимся о том, чтобы обработать это условие правильно.

strictPropertyInitialization

Подобно опции `strictNullChecks`, мы также можем проверить, правильно ли были инициализированы свойства класса, используя флаг компилятора `strictPropertyInitialization`. Рассмотрим следующее определение класса:

```
class WithoutInit {
   a: number;
   b: string;
}
```

Здесь у нас есть стандартное определение для класса `WithoutInit`. У этого класса два свойства: `a` типа `number` и `b` типа `string`. Если опция `strictPropertyInitialization` включена, этот код выдаст ошибки:

```
error TS2564: Property 'a' has no initializer and is not
definitely assigned in the constructor.
error TS2564: Property 'b' has no initializer and is not
definitely assigned in the constructor.
```

Эта ошибка указывает на то, что свойства a и b класса WithoutInit не были инициализированы или присвоены.

Этот класс можно исправить одним из двух способов. Во-первых, можно установить эти значения в конструкторе следующим образом:

```
class WithoutInit {
   a: number;
   b: string;
   constructor(_a: number) {
      this.a = _a;
      this.b = "test";
   }
}
```

Здесь мы определили конструктор для нашего класса, у которого единственный параметр _a типа number. В нашем конструкторе мы устанавливаем свойство класса с именем a для входящего значения _a, а также устанавливаем свойство класса с именем b для жестко закодированного значения "test". Теперь оба свойства правильно инициализированы, и никаких ошибок не будет.

Второй способ исправить этот код – явно сообщить компилятору, что мы знаем, что свойства могут быть неинициализированными:

```
class WithoutInitUndefined {
   a: number | undefined;
   b: string | undefined;
}
```

Здесь у нас есть класс с именем WithoutInitUndefined, у которого два свойства: a и b. Обратите внимание, что свойство a определяется либо как свойство типа number, либо как undefined, а свойство b определяется как string или undefined. Таким образом, мы явно говорим компилятору, что знаем, что a и b могут быть неопределенными, и обработаем этот случай самостоятельно.

noUnusedLocals и noUnusedParameters

При установке опций noUnusedLocals и noUnusedParameters компилятор TypeScript выдаст ошибку, если обнаружит неиспользуемые локальные переменные или неиспользуемые параметры. Рассмотрим следующий код:

```
function testFunction(input: string): boolean {
   let test;
   return false; }
```

Здесь мы определили функцию с именем `testFunction`, которая принимает единственный параметр `input` типа `string`. В теле функции мы определяем переменную с именем `test`, а затем возвращаем булево значение `false`. При компиляции этого кода появятся следующие ошибки:

```
error TS6133: 'input' is declared but its value is never read
error TS6133: 'test' is declared but its value is never read
```

Первая ошибка указывает на параметр функции с именем `input`. Обратите внимание, что он никогда не используется в теле функции, и поэтому не совсем понятно, почему он был объявлен первым.

Вторая ошибка указывает на то, что хотя мы и определили переменную с именем `test` в теле функции, мы нигде ее не использовали. Так почему она была объявлена в первую очередь?

Использование этих параметров компиляции поможет нам обрезать любые неиспользуемые локальные переменные или неиспользуемые параметры функции.

noImplicitReturns

Если функция объявила, что возвратит значение, то компилятор может гарантировать, что она действительно сделает это, используя опцию `noImplicitReturns`. Часто при работе с большой функцией или во время рефакторинга существующей функции мы можем непреднамеренно внести нежелательные недостатки логики, если не будем осторожны. Рассмотрим следующий код:

```
function isLargeNumber(value: number): boolean | undefined {
   if (value > 1_000_000)
      return true;
}
```

Здесь мы определили функцию `isLargeNumber`, которую можно вызывать с одним параметром `value`. Эта функция может возвращать либо логическое, либо неопределенное значение. Обратите внимание, что в общем случае эта функция должна возвращать только логическое значение, но в рамках этого обсуждения мы также допускаем неопределенное. Этот код выдаст следующую ошибку:

```
error TS7030: Not all code paths return a value
```

Причина тому – ошибки в логике в рамках этой функции. Обратите внимание, что есть только оператор `if`, который проверяет, является ли аргумент `value` больше 1 миллиона.

Нет оператора `esle` или даже возврата по умолчанию. Поэтому если аргумент `value` меньше или равен 1 миллиону, мы просто «проваливаемся» сквозь оператор `if` и ничего не делаем.

Однако определение функции указывает, что мы должны возвращать либо логическое значение, либо неопределенное, и в этом случае мы забыли что-либо вернуть. Поэтому компилятор выделил логическую ошибку в нашей функции, которая должна быть исправлена.

noFallthroughCasesInSwitch

Для дальнейшей проверки логических ошибок в нашем коде мы можем использовать опцию `noFallthroughCasesInSwitch` для проверки операторов `Switch`. Рассмотрим следующий код:

```
enum SwitchEnum {
   ONE,
   TWO
}

function testEnumSwitch(value: SwitchEnum): string {
   let returnValue = "";
   switch(value) {
      case SwitchEnum.ONE:
         returnValue = "One";
      case SwitchEnum.TWO:
         returnValue = "Two";
   }
   return returnValue;
}
```

Здесь мы определили перечисление с именем `SwitchEnum`, у которого есть два значения, `ONE` и `TWO`. Затем мы определяем функцию `testEnumSwitch`, которая возвращает строковое значение на основе входного параметра `value` типа `SwitchEnum`. В этой функции мы определяем переменную `returnValue` и устанавливаем для нее пустую строку. Затем у нас идет оператор `switch`, который присваивает значение переменной `returnValue` `"One"` или `"Two"` на основе переданного перечисления. Этот код выдаст следующую ошибку:

```
error TS7029: Fallthrough case in switch.
```

Эта ошибка правильно идентифицирует логическую ошибку в нашем операторе `switch`. Обратите внимание, что если значение входного аргумента – `SwitchEnum.ONE`, мы присваиваем значение переменной `returnValue` `"One"`. К сожалению, логика автоматически проваливается на втором операторе case, в котором переменная `returnValue` установлена в значение `"Two"`. Это означает, что независимо от того, передано в качестве аргумента `value SwitchEnum.ONE` или `SwitchEnum.TWO`, возвращаемое значение всегда будет `"Two"`.

Этот код необходимо исправить:

```
function testEnumSwitch(value: SwitchEnum): string {
   let returnValue = "";
   switch(value) {
      case SwitchEnum.ONE:
         returnValue = "One";
         break;
      case SwitchEnum.TWO:
         returnValue = "Two";
   }
   return returnValue;
}
```

Здесь тонкое отличие, которое исправляет ошибку компиляции, заключается во включении оператора `break` в логику case `SwitchEnum.ONE`. Этот оператор завершает работу оператора `switch` и избегает «проваливания» кода сквозь второй оператор `case`.

Итак, мы увидели, что компилятор правильно определяет наличие явных ошибок в наших операторах `switch` и места, где логика будет «проваливаться» от одной конструкции `switch case` к другой. Хотя может показаться, что эти ошибки легко обнаружить в таких простых примерах, они могут быстро потеряться в больших базах кода.

strictBindCallApply

JavaScript предоставляет функции `bind`, `call` и `apply`, когда нам нужно переопределить значение переменной this внутри функции. Мы уже видели, как используется функция `apply` при обсуждении декораторов в `главе 4` *«Декораторы, обобщения и асинхронные функции»*. Мы подробно обсудим использование функции `bind` в `главе 7` *«Фреймворки, совместимые с TypeScript»*. Опцию `strictBindCallApply` можно использовать как гарантию того, что параметры, используемые в любом из этих вызовов, являются типобезопасными. Чтобы проиллюстрировать эту концепцию, рассмотрим приведенный ниже код:

```
class MyBoundClass {
   name: string = "defaultNameValue";
   printName(index: number, description: string) {
      console.log(`this.name: ${this.name}`);
      console.log(`index: ${index}`);
      console.log(`description: ${description}`);
   }
}

let testBoundClass = new MyBoundClass();
testBoundClass.printName(1, 'testDesc');
```

Здесь мы определили класс `MyBoundClass` с единственным свойством `name`, для которого установлено значение по умолчанию `"defaultNameValue"`. Затем мы определили функцию `printName`, у которой есть два параметра. Первый параметр – `index` типа `number`, второй – `description` типа `string`. Функция `printName` записывает значение `this.name` в консоль, а затем также записывает значение аргументов `index` и `description`. Затем мы создаем экземпляр класса `MyBoundClass` и присваиваем его переменной `testBoundClass`. Наконец, мы вызываем функцию `printName` с двумя аргументами. Вывод этого фрагмента кода выглядит так:

```
this.name : defaultNameValue
index : 1
description : testDesc
```

Здесь видно, что значения, записанные в консоль, соответствуют ожидаемым. В частности, обратите внимание, что значение свойства `this.name` – это то, что мы установили по умолчанию для этого класса. Если мы теперь используем JavaScript-функцию `call`, то можем переопределить свойство `this.name`:

```
testBoundClass.printName.call(
   { name: `overridden name property value` },
   1, 'call: whoah !');
```

Здесь мы выполняем функцию `printName` переменной `testBoundClass`, которая является экземпляром класса `MyBoundClass`. На этот раз, однако, мы используем функцию `call`, использующую первый параметр, который мы предоставляем в качестве значения `this`.

Второй и третий аргументы затем передаются в функцию `printName`. Результаты этого вызова следующие:

```
this.name : overridden name property value
index : 1
description : call : whoah !
```

Здесь видно, что свойство name класса `MyBoundClass` было переопределено со значения по умолчанию `"defaultNameValue"` в значение свойства name из объекта, который был передан в функцию call, которая была `"overridden name property value"`. Другими словами, используя функцию `call`, мы можем переопределить значение свойства `this.name`, указав значение, которое будет использоваться для свойства `this`.

Компилятор TypeScript выдаст ошибки, если типы свойств, использованных при вызове функции `call`, неверны. Это можно проиллюстрировать так:

```
testBoundClass.printName.call(
   { name: `overridden name property value` },
   "string", 12);
```

Здесь мы используем функцию `call`, чтобы переопределить значение `this`, которое используется в классе `MyBoundClass`. К сожалению, мы предоставили строковое значение `"string"` для первого параметра, который должен был иметь тип `number`. В этом случае компилятор TypeScript выдаст следующую ошибку:

```
error TS2345: Argument of type '"string"' is not assignable to
parameter of type 'number', "string", 12);
```

Здесь видно, что компилятор TypeScript фактически генерирует ошибки, если параметры, переданные в функцию, не имеют правильных типов. Он применяет эти правила, даже если мы используем функцию `call`, так как мы указали, что опция `strictBindCallApply` имеет значение `true`.

Обратите внимание, что формат функции `apply` использует массив для указания списка аргументов:

```
testBoundClass.printName.apply(
   { name: `overridden by apply` },
   [1, 'apply: whoah!']);
```

Здесь мы используем JavaScript-функцию `apply`, чтобы переопределить значение `this` внутри экземпляра класса `MyBoundClass`. Однако вместо применения списка параметров мы используем массив аргументов.

Использование опции `strictBindCallApply` приведет к возникновению ошибок, если какая-либо из функций – `bind`, `call` и `apply` – используется с неверными типами параметров.

Резюме

В этой главе мы рассказали о том, что нам нужно знать, чтобы написать и использовать собственные файлы объявлений. Мы обсудили глобальные переменные JavaScript в визуализированном HTML-коде и то, как получить к ним доступ в TypeScript. Затем мы перешли к небольшой вспомогательной JavaScript-функции и написали собственный файл объявлений. После этого мы перечислили несколько правил определения модулей, выделив необходимый синтаксис JavaScript, и показали, каким будет эквивалентный синтаксис объявления в TypeScript.

Во втором разделе мы рассмотрели строгие опции компилятора TypeScript и сравнили код, содержащий каждый из этих параметров, и код без них.

В следующей главе мы рассмотрим, как использовать существующие сторонние библиотеки JavaScript и как импортировать существующие файлы объявлений для этих библиотек в свои проекты TypeScript.

Глава 6

Сторонние библиотеки

Наша среда разработки TypeScript не будет иметь большого значения, если мы не сможем повторно использовать множество существующих библиотек JavaScript, фреймворков и общих преимуществ, которые делают разработку на JavaScript столь популярной сегодня. Однако, как мы видели, для того чтобы использовать конкретную стороннюю библиотеку с TypeScript, нам сначала понадобится соответствующий файл определений.

Вскоре после выпуска TypeScript Борис Янков создал репозиторий GitHub для размещения файлов определений TypeScript для сторонних JS-библиотек. Этот репозиторий под названием

DefinitiveTyped (`https://github.com/borisyankov/DefiniteTyped`) быстро стал очень популярным, и в настоящее время здесь можно найти высококачественные файлы определений.

В настоящее время в DefinitiveTyped находится свыше 5000 файлов определений, созданных с течением времени сотнями участников со всего мира.

Если бы мы оценили успех TypeScript в рамках сообщества JavaScript, то репозиторий DefintelyTyped был бы хорошим показателем того, насколько хорошо был принят TypeScript. Прежде чем продолжить и попытаться написать собственные файлы определений, посетите репозиторий DefinitiveTyped, чтобы увидеть, есть ли там уже такой файл.

В этой главе мы более подробно изучим использование файлов определений и рассмотрим следующие темы:

- использование NuGet в Visual Studio;
- применение `npm` и `@types`;
- выбор JavaScript -фреймворка;
- использование TypeScript с Backbone;
- применение TypeScript с Angular 1;
- использование TypeScript с ExtJS.

Использование файлов определений

Существует несколько способов включить файлы определений в свой проект. Когда изначально был выпущен TypeScript, единственный способ включить фай-

лы определений состоял в том, что их нужно было скачать с DefinitiveTyped, сохранить в каталоге в своем проекте и затем обращаться к ним вручную. По мере того как популярность TypeScript начала расти, количество файлов определений и, следовательно, поиск и установка правильных файлов стали более сложным процессом. Затем стали появляться утилиты, чтобы облегчить работу. Первой из них была утилита командной строки `tsd` на основе Node, которая позволяла запрашивать базу данных DefintelyTyped и устанавливать соответствующие файлы определений. `tsd` устарела в начале 2016 года в пользу другой утилиты под названием Typings. Typings также разрешала запросы и установку файлов определений на основе базы данных DefiniteTyped. Одной из основных функций Typings была возможность нацеливаться на конкретную версию файла определения и сохранять соответствующую информацию в файле конфигурации с именем `typings.json`.

С выходом TypeScript 2.0 возможность установки файлов определений типов была включена в популярный пакет `npm`, и поэтому никаких других инструментов для управления файлами определений не требовалось. Если вы используете Node, `npm` и файл `package.json`, то `npm` обладает всеми возможностями, которые необходимы, чтобы включить файлы определений в свой проект.

Если вы работаете с интегрированными средами разработки Visual Studio или WebMatrix, то можете использовать NuGet для достижения того же результата. В этом разделе мы рассмотрим способы управления файлами определений, предоставляемые NuGet и `npm`.

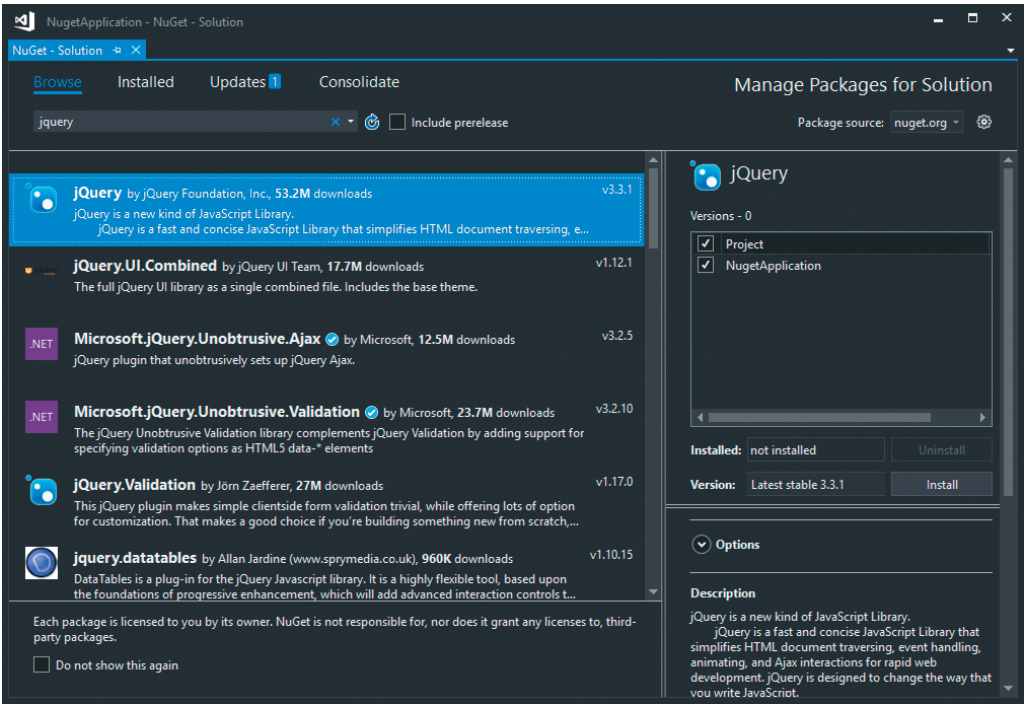
Использование NuGet

NuGet – это популярная платформа управления пакетами, которая скачивает необходимые внешние библиотеки и автоматически включает их в ваш проект Visual Studio или WebMatrix. Она может использоваться для внешних библиотек, которые упакованы в виде DLL, например StructureMap, или для JavaScript-библиотек и файлов объявлений. NuGet также доступна в качестве утилиты командной строки.

Использование диспетчера расширений NuGet

Чтобы использовать диалоговое окно диспетчера пакетов NuGet в Visual Studio, вам необходимо открыть существующий проект. После открытия проекта выберите опцию **Tools** (**Инструменты**) на главной панели инструментов, затем выберите **NuGet Package Manager** (**Диспетчер пакетов NuGet**) и, наконец, **Manage NuGet Packages for Solution**.

Откроется диалоговое окно диспетчера пакетов NuGet. В левой части диалогового окна нажмите **Browse** (**Обзор**). После этого диалоговое окно NuGet запросит веб-сайт NuGet и покажет список доступных пакетов. В левом верхнем углу экрана находится поле поиска. Шелкните мышью внутри него и введите слово `jquery`, чтобы показать все пакеты, доступные в NuGet для jQuery, как показано ниже:

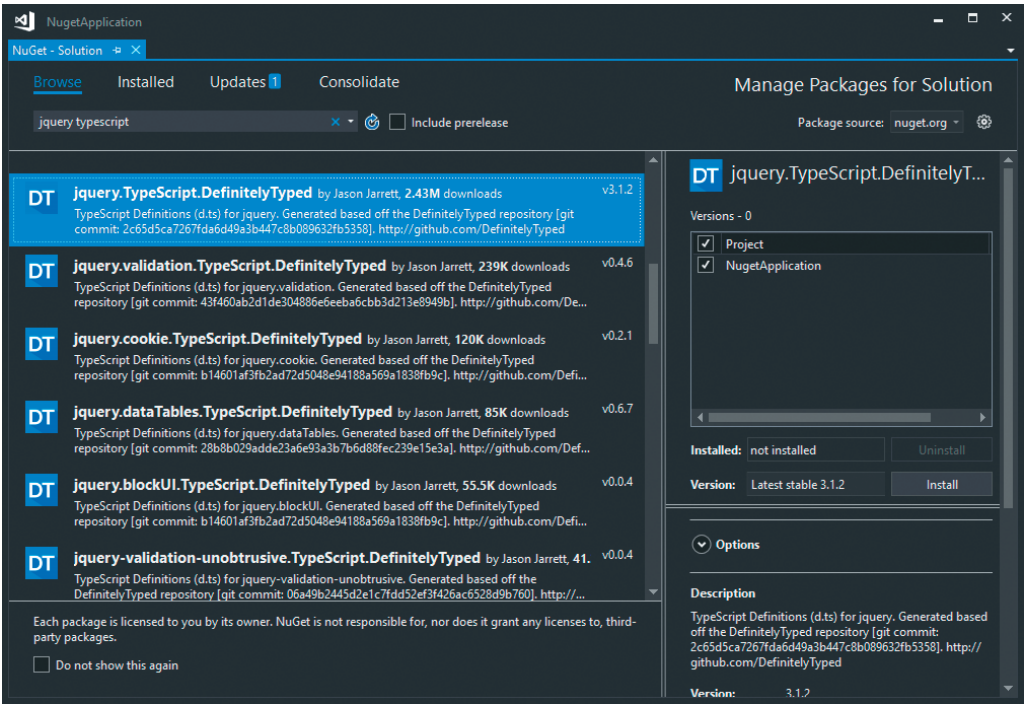


Каждый пакет будет выделен с помощью кнопки **Install** (**Установить**), когда вы будете выбирать его на панели результатов поиска. После выбора пакета на панели справа отобразится более подробная информация о рассматриваемом пакете NuGet. Обратите внимание, что на панели сведений о проекте также показана версия пакета, который вы собираетесь установить. При нажатии на кнопку **Install** (**Установить**) будут загружены соответствующие файлы, а также любые зависимости, которые будут автоматически включены в ваш проект.

Каталог установки, который NuGet использует для файлов JavaScript, на самом деле называется `Scripts`, и это не каталог `lib`, который мы создали ранее. NuGet в качестве стандарта использует каталог `Scripts`, поэтому любые пакеты, содержащие JavaScript, будут устанавливать соответствующие файлы JavaScript в каталог `Scripts`.

Установка файлов объявлений

Вы обнаружите, что у большинства наиболее популярных файлов объявлений, которые находятся в репозитории DefiniteTyped, есть соответствующий пакет NuGet. Эти пакеты называются `<library>.TypeScript.DefinitelyTyped` в качестве стандартного соглашения об именовании. Если мы введем jquery typcript в поле поиска NuGet, то увидим список возвращаемых пакетов DefiniteTyped. Пакет NuGet, который мы ищем, носит название `jquery.TypeScript.DefiniteTyped`. Он был создан Джейсоном Джарретом. Его версия на момент написания этой главы – 3.1.2, как показано ниже:



У пакетов DefiniteTyped есть собственный внутренний номер версии, и эти номера версий не обязательно совпадают с версией используемой вами библиотеки JavaScript.

При установке пакета `jQuery.TypeScript.DefiniteTyped` в каталоге Scripts будет создан каталог typings, а затем включен файл определения `jquery.d.ts`. Данный стандарт именования каталогов был принят авторами пакетов NuGet.

Использование консоли диспетчера пакетов

В Visual Studio также есть версия диспетчера пакетов NuGet для командной строки, доступная в виде консольного приложения. Она тоже интегрирована в Visual Studio. После нажатия на **Tools**, затем **NuGet Package Manager** и, наконец, на **Package Manager Console** (**Консоль диспетчера пакетов**) появится новое окно Visual Studio и будет инициализирован интерфейс командной строки NuGet. У версии NuGet для командной строки есть ряд функций, которые не включены в GUI-версию. Наберите `get-help NuGet`, чтобы увидеть список доступных аргументов командной строки верхнего уровня.

Установка пакетов

Чтобы установить пакет NuGet из командной строки консоли, просто введите `install-package <packageName>`. В качестве примера, чтобы установить пакет `jquery.TypeScript.DefiniteTyped`, просто введите:

```
Install-Package jquery.TypeScript.DefinitelyTyped
```

После этого будет выполнено подключение к серверу NuGet, а пакет будет загружен и установлен в ваш проект.

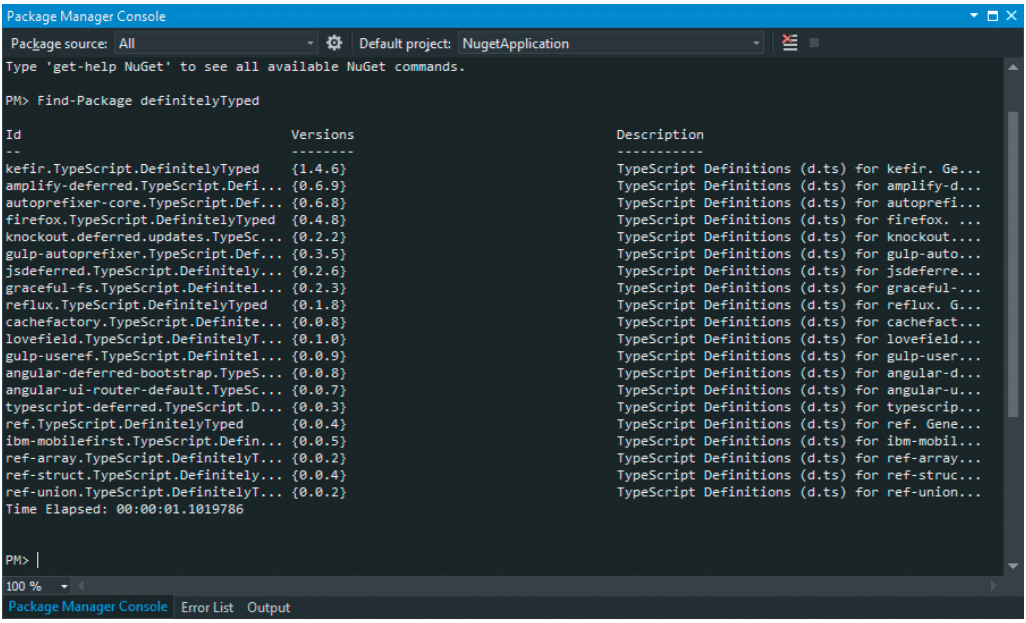
На панели инструментов окна **Package Manager Console** есть два раскрывающихся списка: **Package Source** (**Источник пакета**) и **Default Project** (**Проект по умолчанию**). Если в вашем решении Visual Studio есть несколько проектов, вам нужно выбрать правильный проект для NuGet, в который вы хотите установить пакет, из раскрывающегося списка «**Проект по умолчанию**».

Поиск имен пакетов

Поиск имен пакетов из командной строки выполняется с помощью команды `Find-Package`. Например, чтобы найти доступные пакеты, содержащие строку поиска `definitelytyped`, выполните следующую команду:

```
Find-Package definitelytyped
```

Будут перечислены пакеты, которые соответствуют критериям поиска:



Установка конкретной версии

Существует ряд библиотек JavaScript, которые не совместимы с jQuery версии 2.x, и для них потребуется версия jQuery в диапазоне 1.x. Чтобы установить конкретную версию пакета NuGet, нам нужно указать параметр `-Version` из командной строки. Чтобы установить пакет `jquery v. 1.11.1`, например, запустите из командной строки:

```
Install-Package jQuery -Version 1.11.1
```

NuGet обновит или понизит версию устанавливаемого пакета, если обнаружит, что в вашем проекте уже установлена другая версия. В предыдущем примере мы уже установили последнюю версию jQuery (2.1.1) в своем проекте, поэтому NuGet сначала удалите `jQuery 2.1.1` перед установкой `jQuery 1.11.1`.

Использование npm и @types

С выходом компилятора TypeScript версии 2.0 теперь у нас также есть возможность устанавливать файлы объявлений, используя `npm`. Это означает, что в нашем наборе инструментов нет никакой разницы для установки зависимостей проекта, поскольку он должен включать файлы объявлений. В качестве примера,

чтобы установить библиотеку Underscore в качестве зависимости проекта, мы набрали бы следующее:

```
npm install underscore
```

И чтобы установить файлы объявлений для Underscore, мы можем набрать:

```
npm install @types/underscore
```

Обратите внимание на префикс `@types`, используемый в команде `npm`. Этот специальный синтаксис инструктирует npm устанавливать файлы объявлений для Underscore. Это очень тонкий, но легко запоминающийся синтаксис.

Если мы посмотрим на файл `package.json` в каталоге нашего проекта, то заметим, что библиотеки Underscore и соответствующие библиотеки типов зарегистрированы:

```
{
   "name": "npm_types",
   "version": "1.0.0",
   "description": "",
   "main": "index.js",
   "scripts": {
      "test": "echo \"Error: no test specified\" && exit 1"
   },
   "author": "",
   "license": "ISC",
   "dependencies": {
      "@types/underscore": "^1.8.9",
      "underscore": "^1.9.1"
   }
}
```

Здесь у нас есть две записи в разделе зависимостей файла `package.json`, одна для самого `underscore`, а другая для `@types/underscore`. Обратите внимание, что, как и в случае стандартной зависимости npm, мы автоматически получаем доступ к используемой версии библиотеки.

Этот механизм включения определений типов с помощью npm был принят в качестве стандартного механизма.

Использование сторонних библиотек

В этом разделе мы приступим к изучению ряда старых сторонних библиотек JavaScript, их файлов объявлений и способов написания совместимого кода TypeScript для каждого из этих фреймворков. Мы сравним Backbone, Angular

(версия 1) и Ext JS, которые являются фреймворками для создания полнофункциональных клиентских JavaScript-приложений. В ходе нашего обсуждения мы увидим, что некоторые фреймворки в высшей степени совместимы с языком TypeScript и его функциями, некоторые совместимы частично, а у некоторых совместимость очень низкая.

В следующей главе мы рассмотрим сторонние библиотеки JavaScript, которые были написаны явно с учетом TypeScript или для которых был изменен компилятор TypeScript. Однако в оставшейся части этой главы мы сосредоточимся на стандартных сторонних библиотеках, разработанных для поддержки JavaScript.

Несмотря на то что существует ряд новых фреймворков, которые поддерживают TypeScript из коробки, вы можете обнаружить, что время от времени вам необходимо работать со старыми.

Выбор фреймворка JavaScript

Выбор фреймворка или библиотеки JavaScript для разработки одностраничных приложений – сложная и иногда пугающая задача. Кажется, что каждый месяц появляется новый фреймворк, обещающий все больше и больше функциональности для все меньшего и меньшего количества кода. Чтобы помочь разработчикам сравнить эти фреймворки и сделать осознанный выбор, Эдди Османи написал отличную статью под названием **Путешествие по джунглям JavaScript MVC**, которая доступна по адресу `http://www.smashingmagazine.com/2012/07/27/journey-through-the-javascript-mvc-jungle`.

По сути, его совет прост – это личный выбор каждого, поэтому попробуйте несколько фреймворков и посмотрите, какой из них лучше всего соответствует вашим потребностям, вашему мышлению и имеющимся у вас навыкам программирования. Проект *TodoMVC* (`http://todomvc.com`), который начал Эдди, делает отличную работу, реализуя одно и то же приложение в ряде фреймворков **MV*** (**Model View Something**). В действительности это справочный сайт, который можно использовать для того, чтобы покопаться в полностью работающем приложении и сравнить для себя методы кодирования и стили разных фреймворков.

Опять же, в зависимости от библиотеки JavaScript, которую вы используете в TypeScript, вам может потребоваться написать свой код TypeScript особым образом. При выборе фреймворка имейте это в виду – если его трудно использовать с TypeScript, то вам лучше рассмотреть другой фреймворк, обеспечивающий лучшую интеграцию. Если работать с этим фреймворком в TypeScript легко и непринужденно, то ваша производительность и весь опыт разработки будут намного лучше.

В этом разделе мы рассмотрим ряд старых библиотек JavaScript, а также их файлы объявлений и узнаем, как написать совместимый код TypeScript. Главное, что

нужно помнить, – TypeScript генерирует JavaScript, поэтому если вы боретесь за использование сторонней библиотеки, взломайте сгенерированный JavaScript и посмотрите, как выглядит код, который генерирует TypeScript. Если сгенерированный JavaScript соответствует образцам кода JavaScript в документации библиотеки, то вы на правильном пути. Если нет, то вам может понадобиться изменить свой TypeScript, пока скомпилированный JavaScript не будет совпадать с примерами.

При попытке написать код TypeScript для стороннего JavaScript-фреймворка – особенно если вы работаете с документацией JavaScript – наша первоначальная попытка может быть просто пробой и ошибкой. В оставшейся части этой главы показано, что три разные библиотеки требуют разных способов написания TypeScript.

Backbone

Backbone – это популярная JavaScript-библиотека, которая обеспечивает структуру веб-приложений, предоставляя модели, коллекции и представления. Backbone существует с 2010 года, и у нее очень большое количество поклонников благодаря множеству коммерческих веб-сайтов, использующих этот фреймворк. Согласно статье за октябрь 2013 года на сайте infoworld.com (`https://www.infoworld.com/art article/2612250/property/property/thewesttest-javascript-framework-projects.Html?Page=2`), на тот момент у Backbone было свыше 1600 проектов, связанных с Backbone, на GitHub, которые оцениваются в три звезды, что означает, что он обладает обширной экосистемой расширений и связанных библиотек.

Давайте кратко рассмотрим Backbone, написанный на TypeScript.

Среду Backbone можно настроить с помощью npm:

```
npm install backbone
npm install @types/backbone
```

Использование наследования с Backbone

Из документации к Backbone мы находим пример создания класса `Backbone.Model` в JavaScript:

```
var NoteModel = Backbone.Model.extend ({
   initialize: function() {
      console.log("NoteModel initialized.");
   },
   author: function() {},
```

```
    coordinates : function() {},
    allowedToEdit: function(account) {
       return true;
    }
});
```

Этот код демонстрирует типичное использование Backbone в JavaScript. Вначале мы создаем переменную `NoteModel`, которая расширяет (или наследует от) `Backbone.Model`. Это можно увидеть с помощью синтаксиса `Backbone.Model.extend`. Backbone-функция `extend` использует нотацию объекта для определения объекта внутри фигурных скобок `{...}`. В этом примере у объекта `NoteModel` есть четыре функции: `initialize`, `author`, `coordinates` и `allowToEdit`.

Согласно документации Backbone, функция `initialize` будет вызываться после создания нового экземпляра этого класса. В нашем примере она просто записывает сообщение в консоль, чтобы указать, что функция была вызвана. На этом этапе функции `author` и `coordinates` остаются пустыми, и только функция `allowToEdit` действительно что-то делает – возвращает значение `true`.

Если затем мы щелкнем правой кнопкой мыши по функции `extend` и затем выберем опцию **Go to Definition** (**Перейти к определению**) в коде Visual Studio, то будем перенаправлены в файл определений Backbone. Здесь мы видим интересный момент в документации к этой функции, а именно:

```
class Model extends ModelBase {

/**
* Предпочтительнее использовать ключевое слово TypeScript extend;
**/
public static extend(properties: any, classProperties?: any): any;
```

Этот фрагмент файла объявлений показывает определение класса `Backbone.Model`.

Видно, что функция `extends` определяется как `public static`. Обратите внимание, однако, на комментарий в блоке кода – Предпочтительнее использовать ключевое слово TypeScript `extend`.

Данный комментарий указывает на то, что файл объявлений для Backbone построен на основе ключевого слова TypeScript `extends`, что позволяет использовать естественный синтаксис наследования TypeScript для создания объектов Backbone. Поэтому давайте очистим наш код, чтобы использовать ключевое слово TypeScript `extends` для наследования класса от базового класса `Backbone.Model`:

```
class NoteModel extends Backbone.Model {
   initialize() {
      console.log(`NoteModel initialized`);
   }
```

```
    author() {}
    coordinates() {}
    allowedToEdit(account: any): boolean {
      return true;
    }
}
```

Теперь мы создаем определение класса с именем `NoteModel`, которое использует синтаксис наследования TypeScript для наследования класса от базового класса `Backbone.Model`. У этого класса затем появляются функции `initialize`, `author`, `coordinates` и `allowToEdit`, аналогичные тем, что мы видели в предыдущей версии кода JavaScript.

Давайте теперь создадим экземпляр объекта `NoteModel`, создав HTML-страницу:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title></title>
  <script src="./node_modules/underscore/underscore.js">
  </script>
  <script src="./node_modules/backbone/backbone.js"></script>
  <script src="./backbone_app.js"></script>
</head>
<body>
  <script >
    window.onload = function() {
      console.log('window.onload called');
      var noteModel = new NoteModel();
    };
  </script>
</body>
</html>
```

Здесь у нас есть стандартная HTML-страница, которая включает в себя как файл `backbone.js`, так и файл `underscore.js` (Backbone зависит от Underscore). Затем мы включаем наш файл `backbone_app.js`, который содержит определение класса `NoteModel`. Далее идет тег `<script>`, который будет выполнять функцию при загрузке страницы. В рамках этой функции мы создаем экземпляр класса `NoteModel`. Согласно документации, функция `initialize` класса `Backbone.Model` будет вызываться Backbone при создании экземпляра класса `Model`. Проверяя вывод консоли на этой странице, мы увидим следующие сообщения:

```
window.onload called
NoteModel initialized
```

Видно, что функция `window.onload` действительно была вызвана и что мы успешно создали экземпляр `Backbone.Model`.

Все основные объекты Backbone разработаны с учетом наследования. Это означает, что при создании новых коллекций, представлений и маршрутизаторов Backbone будет использоваться тот же синтаксис `extends` в TypeScript. Следовательно, Backbone очень хорошо подходит для TypeScript, потому что мы можем использовать естественный синтаксис наследования TypeScript для создания новых объектов Backbone.

Использование интерфейсов

Поскольку Backbone позволяет использовать наследование TypeScript для создания объектов, мы можем также легко использовать интерфейсы TypeScript с любым из наших объектов Backbone. Извлечение интерфейса для класса `NoteModel` из предыдущего примера будет выглядеть так:

```
interface INoteModel {
   initialize(): void;
   author(): void;
   coordinates(): void;
   allowedToEdit(account: any): boolean;
}
```

Теперь мы можем обновить наше определение класса `NoteModel` для реализации этого интерфейса:

```
class NoteModel extends Backbone.Model implements INoteModel {
   // Существующий код;
}
```

Наше определение класса теперь реализует интерфейс `INoteModel` TypeScript. Это простое изменение защищает наш код от непреднамеренного изменения, а также открывает возможность для работы с базовыми объектами Backbone в стандартных объектно-ориентированных шаблонах проектирования. В случае необходимости мы могли бы применить *шаблон Factory*, описанный в `главе 3` *«Интерфейсы, классы и наследование»*, для возврата определенного типа модели Backbone или любого другого объекта Backbone.

Использование синтаксиса обобщений

Файл объявлений Backbone также добавил синтаксис обобщений в некоторые определения классов. Это дает дополнительные преимущества при написании кода TypeScript для Backbone. Коллекции Backbone (сюрприз, сюрприз) со-

держат коллекцию моделей Backbone, что позволяет нам определять коллекции в TypeScript:

```
class NoteCollection extends Backbone.Collection<NoteModel> {
   model = NoteModel;
}
```

Здесь у нас есть коллекция `NoteCollection`, которая наследует от `Backbone.Collection`, а также использует синтаксис обобщений, чтобы ограничить коллекцию обработкой только объектов типа `NoteModel`. Это означает, что любая из стандартных функций коллекции, таких как `at()` или `pluck()`, будет строго типизирована для возврата моделей `NoteModel`, что еще больше повысит нашу безопасность типов и улучшит технологию Intellisense.

Обратите внимание на синтаксис, используемый для присваивания типа внутреннему свойству `model` класса коллекции во второй строке. Мы не можем использовать стандартный синтаксис TypeScript: `model = NoteModel`, так как это приводит к ошибке времени компиляции. Нам нужно присвоить свойство model определению класса, как видно из синтаксиса `model = NoteModel`.

Использование ECMAScript 5

Backbone также позволяет использовать возможности ECMAScript 5 для определения получающих и устанавливающих методов классов `Backbone.Model`:

```
interface ISimpleModel {
   name: string;
   id: number;
}

class SimpleModel extends Backbone.Model implements ISimpleModel {
   get name() {
      return this.get('name');
   }
   set name(value: string) {
      this.set('name', value);
   }
   get id() : number {
      return this.get('id');
   }
   set id(value: number) {
      this.set('id', value);
   }
}
```

Здесь мы определили интерфейс с двумя свойствами, `ISimpleModel`. Затем мы определяем класс `SimpleModel`, который наследует от `Backbone.Model`, а также реализует интерфейс `ISimpleModel`. Далее идут получающие и устанавливающие методы ES5 для свойств `id` и `name`. Backbone использует атрибуты класса для хранения значений модели, поэтому получающие и устанавливающие методы просто вызывают основные методы `get` и `set` `Backbone.Model`.

Совместимость Backbone с TypeScript

Как мы уже видели, Backbone позволяет использовать все возможности языка TypeScript в своем коде. Мы можем использовать классы, интерфейсы, наследование, обобщения и даже свойства ECMAScript 5. Все наши классы также наследуют от базовых объектов Backbone. Это делает Backbone высокосовместимой библиотекой для создания веб-приложений с использованием TypeScript. Мы подробнее рассмотрим библиотеку Backbone в следующих главах.

Angular 1

AngularJs версии 1 (или просто Angular 1) – очень популярный фреймворк JavaScript. Он был разработан и распространялся Google, но позже был заменен на Angular 2, который использует TypeScript в качестве выбранного языка. Мы рассмотрим его в следующей главе. В этом разделе будет обсуждаться написание кода Angular 1 с TypeScript в качестве примера полусовместимой сторонней библиотеки.

Angular 1 использует совершенно другой подход к созданию JavaScript SPA, применяя HTML-синтаксис, который понимает работающее приложение Angular 1. Это дает приложению возможности для двусторонней привязки данных, благодаря чему автоматически синхронизируются модели, представления и HTML-страница. Angular 1 также предоставляет механизм внедрения зависимости и использует службы для предоставления данных вашим представлениям и моделям.

Давайте рассмотрим пример из руководства к Angular v. 1.7 (`https://docs.Angularjs.org/tutorial /step_02`), обнаруженный во втором шаге, где мы приступаем к созданию контроллера с именем `PhoneListController`. В примере, приведенном в руководстве, показан следующий код:

```
var phonecatApp = angular.module('phonecatApp', []);

phonecatApp.controller('PhoneListController', function ($scope)
{
   $scope.phones = [
     {'name': 'Nexus S',
     'snippet': 'Fast just got faster with Nexus S.'},
```

```
      {'name': 'Motorola XOOM with Wi-Fi',
      'snippet': 'The Next, Next Generation tablet.'},
      {'name': 'MOTOROLA XOOM',
      'snippet': 'The Next, Next Generation tablet.'}
    ];
});
```

Этот фрагмент кода типичен для JavaScript-синтаксиса Angular 1. Вначале мы создаем переменную `phonecatApp` и регистрируем ее как модуль Angular, вызывая функцию `module` в глобальном экземпляре `angular`. Первый аргумент этой функции – глобальное имя модуля `Angular`, а пустой массив – это заполнитель для других модулей, которые будут внедрены через процедуры внедрения зависимости Angular 1.

Затем мы вызываем функцию `controller` для только что созданной переменной `phonecatApp` с двумя аргументами. Первый аргумент – это глобальное имя контроллера, а второй – это функция, которая принимает специально именованную переменную Angular `$scope`.

В рамках этой функции код устанавливает в качестве объекта `phones` переменной `$scope` массив объектов JSON, у каждого из которых есть свойство `name` и `snippet`.

Если мы продолжим чтение руководства, то найдем модульный тест, который показывает, как используется контроллер `PhoneListController`:

```
describe('PhoneListController', function(){
  it('should create "phones" model with 3 phones', function() {
    var scope = {},
    ctrl = new PhoneListController(scope);
    expect(scope.phones.length).toBe(3);
  });
});
```

Первые две строки этого фрагмента кода используют глобальную функцию под названием `describe`, а внутри этой функции – другую функцию под названием `it`. Эти две функции являются частью фреймворка модульного тестирования Jasmine. Мы рассмотрим *модульное тестирование* в `главе 8` *«Разработка через тестирование»*, а пока давайте сосредоточимся на оставшейся части кода.

Мы объявляем переменную с именем `scope` как пустой объект JavaScript, а затем переменную `ctrl`, которая использует ключевое слово `new` для создания экземпляра нашего класса `PhoneListController`. Синтаксис `new PhoneListController(scope)` показывает, что Angular 1 использует определение контроллера, так же как мы использовали бы обычный класс в TypeScript.

Создание одного и того же объекта в TypeScript позволило бы нам использовать классы TypeScript, как показано ниже:

```
var phonecatApp = angular.module('phonecatApp', []);
class PhoneListController {
   constructor($scope: any) {
      $scope.phones = [
         {'name': 'Nexus S',
         'snippet': 'Fast just got faster'},
         {'name': 'Motorola',
         'snippet': 'Next generation tablet'},
         {'name': 'Motorola Xoom',
         'snippet': 'Next, next generation tablet'}
      ];
   }
};
```

Наша первая строка такая же, как в нашем предыдущем примере JavaScript. Затем мы используем синтаксис класса TypeScript для создания класса `PhoneListController`. Создав класс TypeScript, мы теперь можем использовать его, как показано в нашем коде теста Jasmine – `ctrl = new PhoneListController(scope)`. Функция конструктора класса `PhoneListController` теперь действует как анонимная функция, которую мы видели в исходном примере JavaScript:

```
phonecatApp.controller('PhoneListController', function ($scope)
{
   // Эта функция замещается конструктором
}
```

Классы Angular и $scope

Давайте немного расширим наш класс `PhoneListController` и посмотрим, как он будет выглядеть по завершении:

```
class PhoneListCtrl {
   myScope: IScope;
   constructor($scope: any, $http: ng.IHttpService, Phone: any) {
      this.myScope = $scope;
      this.myScope.phones = Phone.query();
      $scope.orderProp = 'age';
      _.bindAll(this, 'GetPhonesSuccess');
   }
   GetPhonesSuccess(data: any) {
      this.myScope.phones = data;
   }
};
```

Первое, на что нужно обратить внимание в этом классе, – это то, что мы определяем переменную myScope и сохраняем аргумент $scope, переданный через конструктор, в эту внутреннюю переменную, опять-таки из-за лексических правил видимости JavaScript. Обратите внимание на вызов _.bindAll в конце конструктора – эта служебная функция Underscore гарантирует, что всякий раз, когда вызывается функция GetPhonesSuccess, она будет использовать переменную this в контексте экземпляра класса, а не в контексте вызывающего кода. Мы подробно обсудим использование _.bindAll в следующей главе.

Функция GetPhonesSuccess применяет переменную this.myScope в рамках своей реализации. Вот почему нам нужно было сохранить начальный аргумент $scope во внутренней переменной.

Еще одна вещь, которую мы замечаем, глядя на этот код, заключается в том, что переменная myScope типизируется для интерфейса IScope, который необходимо будет определить следующим образом:

```
interface IScope {
   phones: IPhone[];
}

interface IPhone {
   name: string;
   snippet: string;
}
```

Интерфейс IScope просто содержит массив объектов типа IPhone (простите за неудачное название этого интерфейса – он также может содержать телефоны Android).

Это означает, что у нас нет стандартного интерфейса или типа TypeScript для использования при работе с объектами $scope. По своей природе аргумент $scope будет изменять свой тип в зависимости от того, когда и где его вызывает среда исполнения Angular, поэтому нам необходимо определить интерфейс IScope и строго типизировать переменную myScope для этого интерфейса.

Еще один интересный момент, который стоит отметить в функции конструктора класса PhoneListController, – это тип аргумента $http – ng.IHttpService. Интерфейс IHttpService находится в файле объявлений Angular 1. Чтобы использовать переменные Angular 1, такие как $scope или $http с Typescript, нам нужно найти соответствующий интерфейс в нашем файле объявлений, прежде чем мы сможем использовать любую из функций Angular 1, доступную для этих переменных.

Последнее, на что следует обратить внимание в этом коде конструктора, – это последний аргумент с именем Phone типа any. Давайте кратко рассмотрим реализацию этой службы:

```
var phonecatServices =
   angular.module('phonecatServices', ['ngResource']);
phonecatServices.factory('Phone',
   [
      '$resource', ($resource) => {
        return $resource('phones/:phoneId.json', {}, {
           query: {
              method: 'GET',
              params: {
                 phoneId: 'phones'
              },
              isArray: true
           }
        });
      }
   ]
);
```

В первой строке этого фрагмента кода снова создается глобальная переменная с именем `phonecatServices`, с использованием глобальной функции `angular.module`. Затем мы вызываем функцию `factory`, доступную для переменной `phonecatServices`, чтобы определить наш ресурс `Phone`. Функция `factory` использует строку с именем `'Phone'` для определения ресурса `Phone`, а затем применяет синтаксис внедрения зависимости Angular 1 для внедрения объекта `$resource`. Просматривая этот код, мы видим, что не можем с легкостью создать стандартные классы TypeScript для использования в Angular 1 и не можем использовать стандартные интерфейсы TypeScript или наследование для классов Angular 1.

Совместимость Angular 1 с TypeScript

При написании кода Angular 1 с использованием TypeScript мы можем применять классы в определенных случаях, но должны полагаться на базовые функции Angular 1, такие как `module` и `factory`, для определения наших объектов в других случаях. Кроме того, при использовании стандартных служб Angular 1, таких как `$http` или `$resource`, нам потребуется указать соответствующий интерфейс файла объявлений, чтобы использовать эти службы. Поэтому мы можем описать библиотеку Angular 1 как библиотеку, которая обладает средней совместимостью с TypeScript.

Наследование – Angular 1 против Backbone

Наследование – очень мощная особенность объектно-ориентированного программирования, а также фундаментальная концепция при использовании

JavaScript-фреймворков. Использование контроллера Backbone или контроллера Angular 1 в рамках фреймворка зависит от доступности определенных характеристик или функций. Мы видели, однако, что каждый фреймворк реализует наследование по-своему.

Поскольку JavaScript (до EC6) не обладал концепцией наследования, каждый из этих старых фреймворков должен был найти способ реализовать его. В Backbone реализация наследования осуществляется через функцию `extend` каждого объекта Backbone. Как мы уже видели, ключевое слово TypeScript `extends` следует аналогичной реализации, позволяя фреймворку и языку согласовываться друг с другом.

Angular 1, с другой стороны, использует собственную реализацию наследования и определяет функции в глобальном пространстве имен Angular 1 для создания классов (то есть `angular.module`).

Мы также можем иногда использовать экземпляр приложения (т. е. `<appName>.controller`) для создания модулей или контроллеров. Однако мы обнаружили, что Angular 1 применяет контроллеры способом, который очень напоминает классы TypeScript, и поэтому можем просто создать стандартные классы TypeScript, которые будут работать в приложении Angular 1.

До сих пор мы рассмотрели только синтаксисы Angular 1/TypeScript и Backbone/TypeScript. Цель этого упражнения состояла в том, чтобы попытаться понять, как можно использовать TypeScript в каждом из этих двух сторонних фреймворков.

Обязательно посетите сайт `http://todomvc.com` и ознакомьтесь с полным исходным кодом приложения Todo, написанным на TypeScript для Angular 1 и Backbone. Их можно найти на вкладке **Compile-to-JS** в разделе примеров. Эти работающие примеры кода в сочетании с документацией на каждом из этих сайтов станут бесценным ресурсом при попытке написать синтаксис TypeScript с помощью более старой внешней сторонней библиотеки, такой как Angular 1 или Backbone.

ExtJS

ExtJS – это популярная библиотека JavaScript, обладающая широким спектром виджетов, сеток, компонентов диаграмм и макетов и многого другого. С выпуском 4.0 ExtJS внедрил в свои библиотеки шаблон MVC для архитектуры приложений. Хотя эта бесплатная библиотека для разработки с открытым исходным кодом ExtJS требует лицензию для коммерческого использования. Она популярна среди групп разработчиков, которые разрабатывают замены настольных ПК с поддержкой интернета, поскольку её внешний вид сопоставим с обычными настольными приложениями. По умолчанию ExtJS гарантирует, что каждое приложение или компонент будет выглядеть и чувствовать себя одинаково, независимо от того, в каком браузере они работают, и для нее практически не требуется CSS или HTML.

Команда ExtJS, однако, не выпустила официальный файл объявлений TypeScript для ExtJS, несмотря на большое давление со стороны сообщества. К счастью, на помощь пришло более широкое сообщество TypeScript, начиная с *Майка Обери*. Он написал небольшую утилиту для генерации файлов объявлений из документации ExtJS (`https://github.com/zz9pa/extjsTypescript`).

Повлияла ли эта работа на текущую версию определений ExtJS в DefinitiveTyped, еще неизвестно, но исходные определения от Майка Обери и текущая версия от brian428 в DefiniteTyped очень похожи.

Создание классов в ExtJS

ExtJS – это библиотека JavaScript, которая работает по-своему. Если бы мы классифицировали Backbone, Angular 1 и ExtJS, можно было бы сказать, что Backbone – это высокосовместимая с TypeScript библиотека. Другими словами, языковые особенности классов и наследования внутри TypeScript высокосовместимы с Backbone. В этом случае Angular 1 будет частично совместимой библиотекой с некоторыми элементами объектов Angular 1, соответствующими особенностям языка TypeScript. С другой стороны, ExtJS была бы минимально совместимой библиотекой, с небольшими возможностями языка TypeScript, применимыми к библиотеке, или их отсутствием.

Давайте рассмотрим пример приложения ExtJS 4.0, написанного на TypeScript. Рассмотрим следующий код:

```
Ext.application(
   {
      name: 'SampleApp',
      appFolder: '/code/sample',
      controllers: ['SampleController'],
      launch: () => {
         Ext.create('Ext.container.Viewport', {
            layout: 'fit',
            items: [{
               xtype: 'panel',
               title: 'Sample App',
               html: 'This is a Sample Viewport'
            }]
         });
      }
   }
);
```

Вначале мы создаем приложение ExtJS, вызывая функцию `application` для глобального экземпляра `Ext`. Затем эта функция использует объект JavaScript между первой и последней фигурными скобками `{}` для определения свойств и функ-

ций. Это приложение устанавливает для свойства `name` значение `SampleApp`, для свойства `appFolder` значение `/code/sample`, а для свойства `controllers` – массив с единственной записью – `'SampleController'`.

Потом мы определяем свойство `launch`, которое является анонимной функцией. Затем эта функция использует функцию `create` для создания класса. Функция `create` применяет имя `'Ext.container.Viewport'` для создания экземпляра класса `Ext.container.Viewport`, у которого есть свойства `layout` и `items`. Свойство `layout` может содержать только один определенный набор значений, например `'fit'`, `'auto'` или `'table'`. Массив `items` содержит конкретные объекты ExtJS, которые создаются в зависимости от того, что предлагает их свойство `xtype`.

ExtJS – одна из тех библиотек, которая не интуитивна. Будучи программистом, вам придется постоянно держать окно браузера с документацией по библиотеке открытым и использовать его, чтобы выяснить, что означает каждое свойство для каждого типа доступного класса. В ней также есть много волшебных строк – в предыдущем примере функция `Ext.create` не работала бы, если бы мы неправильно набрали строку `'Ext.container.Viewport'` или просто забыли использовать заглавные буквы в нужных местах. Для ExtJS `'viewport'` и `'ViewPort'` не одно и то же. Помните, что одним из наших решений для волшебных строк в TypeScript является использование перечислений. К сожалению, в текущей версии файла объявлений ExtJS нет набора перечислений для этих типов классов.

Использование приведения типов

Однако мы можем использовать свойство TypeScript под названием приведение типов, чтобы облегчить написание кода ExtJS. Если мы знаем, какой тип объекта ExtJS пытаемся создать, то можем привести объект JavaScript к этому типу, а затем использовать TypeScript, чтобы проверить, правильные ли свойства мы используем для этого типа объекта ExtJS. Чтобы разобраться с этой концепцией, давайте просто примем в расчет внешнее определение `Ext.application`. Без внутреннего кода вызов функции `application` для глобального объекта Ext будет сведен к следующему:

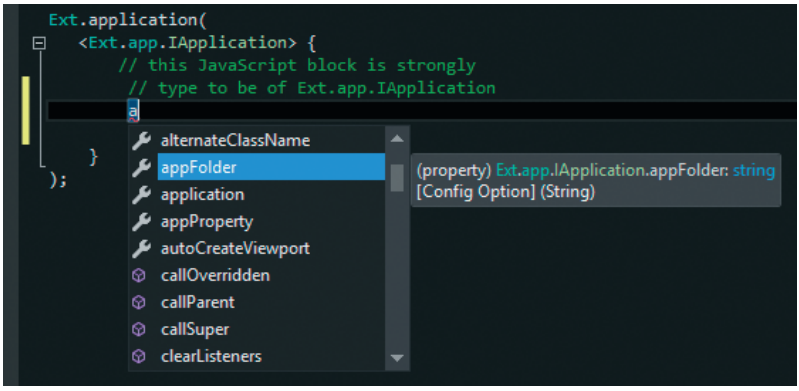
```
Ext.application(
   {
      // Свойства устанавливаются в рамках этого блока
      // кода объекта JavaScript;
   }
);
```

Используя файлы объявлений TypeScript, приведение типов и разумный объем документации по ExtJS, мы знаем, что внутренний объект JavaScript должен иметь тип `Ext.app.IApplication`, и поэтому мы можем привести этот объект

следующим образом:

```
Ext.application(
   <Ext.app.IApplication> {
      // Этот блок строго типизируется, чтобы быть
      // приведенным к типу Ext.app.IApplication;
   }
);
```

Во второй строке данного фрагмента кода теперь используется синтаксис приведения типов TypeScript для приведения объекта JavaScript между фигурными скобками `{}` к типу `Ext.app.IApplication`. Это дает нам строгую проверку типов и Intellisense, как показано ниже:



Аналогичным образом эти явные приведения типов можно использовать для любого объекта JavaScript, который используется для создания классов ExtJS. Файл объявлений для ExtJS, который в настоящее время находится в DefiniteTyped, использует те же имена для своих определений объектов, что и документация по ExtJS, поэтому найти правильный тип должно быть довольно просто.

Предыдущий метод использования явного приведения типов – это почти единственная особенность TypeScript, которую можно использовать с библиотекой ExtJS, но это все еще подчеркивает, как сильная типизация объектов может помочь нам при разработке, делая наш код более надежным и устойчивым к ошибкам.

Компилятор TypeScript специально для ExtJS

Если вы используете ExtJS на регулярной основе, возможно, вам будет интересно взглянуть на работу, проделанную *Гаретом Смитом, Фабиу Парра душ Сантушем* и их командой, на странице `https://github.com/fabioparra/TypeScript`, являющуюся ответвленным проектом компилятора TypeScript, который будет ге-

нерировать классы ExtJS из стандартных классов TypeScript. Использование этой версии компилятора превращает таблицы в обычную разработку на ExtJS, позволяя использовать естественный синтаксис класса TypeScript, наследование через ключевое слово extends, а также естественное именование модулей без необходимости применения волшебных строк. Работа, проделанная этой командой, показывает, что поскольку компилятор TypeScript – это программное обеспечение с открытым исходным кодом, он может быть расширен и изменен для генерации JavaScript определенным образом или для определенной библиотеки. Снимаю шляпу перед *Гаретом*, *Фабиу* и их командой за новаторскую работу в этой области.

Резюме

В этой главе мы рассмотрели сторонние библиотеки JavaScript и то, как их можно использовать в приложении TypeScript. Мы начали с рассмотрения различных способов включения выпущенных сообществом версий файлов объявлений TypeScript в наши проекты с помощью менеджеров пакетов, таких как NuGet и `npm`. Затем мы рассмотрели три вида сторонних библиотек и обсудили, как интегрировать эти библиотеки с TypeScript. Мы изучили Backbone, который можно отнести к категории сторонних библиотек с высокой степенью совместимости; Angular 1, которая является частично совместимой библиотекой; и ExtJS, которая минимально совместима с TypeScript.

Мы увидели, как различные свойства языка TypeScript могут сосуществовать с этими библиотеками, и показали, как будет выглядеть эквивалентный код TypeScript в каждом из этих случаев. В следующей главе мы рассмотрим сторонние библиотеки, предназначенные специально для TypeScript, которые либо создаются с TypeScript, либо обладают полной интеграцией с TypeScript.

Глава 7

Фреймворки, совместимые с TypeScript

Один из переломных моментов в истории TypeScript произошел, когда было объявлено, что команды Microsoft и Google работают вместе над Angular 2. Angular 2 был долгожданным обновлением популярного фреймворка Angular (или Angular 1). К сожалению, это обновление потребовало нового набора языковых функций, чтобы сделать синтаксис Angular 2 более чистым и понятным. Первоначально Google предложил новый язык под названием AtScript, чтобы упростить эти новые языковые функции, которые также были тесно связаны с предложениями ECMAScript 6 и 7.

После нескольких месяцев сотрудничества было объявлено, что все необходимые функции языка AtScript будут включены в TypeScript и что Angular 2 будет написан на TypeScript. Это означало, что поставщики новых языковых функций (TypeScript и Microsoft) и потребители новых языковых функций (Angular 2 и Google) смогли договориться о требованиях и ближайшем будущем языка. Это сотрудничество показывает, что язык TypeScript подвергся тщательному изучению со стороны хорошо известной команды разработчиков JavaScript и прошел долгий путь.

Однако Angular 2 был не первым фреймворком, который принял язык TypeScript. Многие сторонние библиотеки JavaScript также предлагают полную поддержку TypeScript.

В этой главе мы рассмотрим некоторые из этих более популярных фреймворков JavaScript, которые полностью интегрированы в язык TypeScript. Мы сравним синтаксис, используемый в каждом из них, создав один и тот же пример MVC-приложения с использованием данных фреймворков. Сделав это, мы проведем параллельное сравнение, которое покажет нам, как каждый из этих фреймворков решает одни и те же проблемы проектирования. Прежде чем приступить, мы начнем с общего обсуждения того, что такое MVC-фреймворк и как он может помочь нам при разработке.

Мы рассмотрим следующие темы:

- что такое MVC-фреймворк;
- преимущества использования MVC-фреймворка;

- план нашего приложения;
- использование Backbone;
- применение Aurelia;
- использование Angular 2;
- применение ReactJs;
- анализ производительности визуализации.

Что такое MVC?

Акроним MVC расшифровывается как Model-View-Controller (модель–представление–контроллер). Это шаблон проектирования, который помогает в разработке и реализации пользовательских интерфейсов. Пользовательские интерфейсы по своей сути управляются событиями – другими словами: мы отображаем что-то на экране, а затем ждем, пока пользователь сделает что-то, что сгенерирует какое-либо событие. Это может быть отображение графика, или скрытие панели, или выход из нашего приложения. К сожалению, нельзя полностью предопределить точную последовательность событий, которой будет следовать пользователь нашего приложения. Именно эта основанная на событиях парадигма делает проектирование и программирование пользовательского интерфейса гораздо более сложным, чем программа, которая следует определенной последовательности шагов.

Другая сложность пользовательских интерфейсов состоит в том, чтобы попытаться сделать компоненты повторно используемыми. Это означает, что один компонент – такой как строка меню, например – должен иметь возможность использоваться повторно на нескольких страницах. Повторное использование компонентов предлагает свой собственный набор проблем, которые обычно определяют, какой компонент за реакцию на какие события отвечает и как компоненты взаимодействуют, когда многие из них заинтересованы в одном и том же событии.

Шаблон проектирования Model-View-Controller разделяет обязанности пользовательского интерфейса на три основных компонента:

Модель

Модель в MVC предоставляет данные. Как правило, это очень простой **POJO** (**Plain Old JavaScript Object** – старый добрый Java-объект), у которого есть определенные свойства. В качестве примера модели рассмотрим следующий класс TypeScript:

```
interface IModel {
   DisplayName: string;
   Id: number;
}
```

```
class Model implements IModel {
   DisplayName: string;
   Id: number;
   constructor(model : IModel) {
      this.DisplayName = model.DisplayName;
      this.Id = model.Id;
   }
}

let firstModel = new Model({Id: 1, DisplayName: 'firstModel'});
```

Здесь мы определили интерфейс с именем `IModel`, у которого есть свойства `Id` и `DisplayName`, и класс, реализующий данный интерфейс. Мы предоставили простой конструктор для установки этих свойств. Последняя строка этого фрагмента создает экземпляр этого класса с желаемыми свойствами.

Как видно из данного фрагмента, класс Model – это очень простой POJO, который содержит некие данные.

Модели часто содержат другие модели, создавая вложенную структуру информации. Эти модели нередко отображаются непосредственно в структуры, которые возвращаются в формате JSON от конечных точек REST.

Представление

Представление в MVC дает визуальное представление модели. В веб-фреймворках это обычно будет фрагмент HTML-кода:

```
<div id="viewTemplate">
   <span><b> {Id} </b></span>
   <span><h1> {DisplayName} </h1></span>
</div>
```

Здесь у нас есть тег `div`, который содержит два тега `span`. Содержимое первого тега `span` выделено жирным шрифтом и будет отображать свойство `Id` из модели. Содержимое второго тега `span` – заголовок `h1` и отображает свойство `DisplayName` из модели.

Отделяя элементы представления пользовательского интерфейса от модели, мы видим, что можем изменять представление так, как нам нравится, даже не касаясь кода модели. Мы можем применить стили к каждому элементу с помощью CSS или даже полностью скрыть определенные свойства в представлении.

Такое разделение дает нам возможность проектировать или модифицировать часть дисплея независимо от модели. Эта работа по проектированию может быть даже передана совершенно отдельной и независимой команде, обладающей специальными навыками в области проектирования пользовательских интер-

фейсов. Пока базовая модель не изменится, обе части модели и представления будут работать вместе без проблем.

В качестве примера представления рассмотрим следующий код:

```
class View {
   template: string;
   constructor(_template: string) {
      this.template = _template;
   }
   render(model: Model) {
      // Объединяем шаблон и представление;
   }
}
```

Здесь мы определили класс `View`, у которого есть единственное свойство `template`. Когда мы создаем это представление, то даем ему HTML-шаблон, который он должен использовать. У этого класса `View` также есть метод `render` с единственным аргументом `model`. Метод `render` объединит шаблон и модель и вернет конечный HTML-код.

Предыдущие примеры представляют собой псевдокод, предназначенный исключительно для иллюстративных целей, и не являются примером какого-либо MVC-фреймворка.

Контроллер

Контроллер в MVC-фреймворке выполняет работу по координации взаимодействия между моделью и представлением. Контроллер обычно выполняет следующие шаги:

1) создает экземпляр модели;
2) создает экземпляр представления;
3) передает экземпляр модели в представление;
4) просит представление визуализировать себя (сгенерировать фактический HTML-код на основе значений в модели);
5) присоединяет полученный HTML-код к дереву DOM.

Контроллер в MVC также отвечает за логику приложения. Это означает, что он может контролировать, какие представления представлены, когда и что делать, когда происходят определенные события.

В качестве примера того, как может выглядеть контроллер, рассмотрим следующий код:

```
class Controller {
   model: Model;
```

```
    view : View;
    constructor() {
       this.model = new Model({Id : 1, DisplayName : 'firstModel'});
       this.view = new View($('#viewTemplate').html());
    }
    render() {
       $('#domElement').html(this.view.render(this.model));
    }
}
```

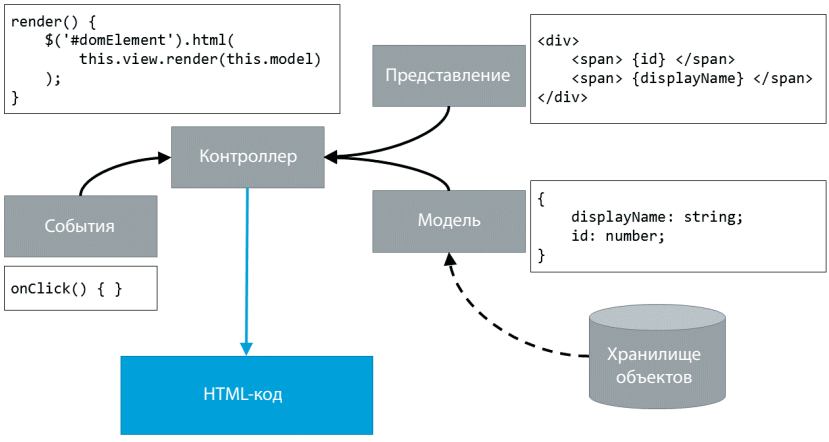
Здесь мы определили класс `Controller`, у которого есть свойства `model` и `view`. Затем наша функция-конструктор создает экземпляр класса `Model` с определенными свойствами и экземпляр класса `View`. Экземпляр класса `View` создается с помощью шаблона, который читается из DOM-элемента `viewTemplate`.

Класс `Controller` также определяет функцию `render`, которая устанавливает фактический HTML-код DOM-элемента `domElement`. Этот HTML-код является результатом вызова функции render для класса `View` и передачи model для визуализации.

Опять же, этот псевдокод используется для описания обязанностей контроллера в MVC. Это не фактический пример конкретного MVC-фреймворка.

Резюмируя

На приведенной ниже диаграмме показаны три элемента шаблона проектирования MVC:



Модели представляют собой простые объекты, содержащие свойства, и обычно создаются из хранилища объектов или базы данных.

Представления – это визуальные HTML-элементы, включающие свойства модели в свои шаблоны.

Контроллеры объединяют модели и представления, отвечают на события и организуют генерацию конечного HTML-кода.

Преимущества использования MVC

Использование MVC-фреймворка дает ряд ощутимых преимуществ, а именно:

- разделение различных элементов, используемых для отображения информации для пользователя;
- повышенную гибкость и повторное использование;
- у одной модели может быть несколько разных представлений, которые могут использоваться в разное время;
- деятельность по разработке пользовательского интерфейса может быть предпринята группой специалистов;
- изменения в данных модели могут инициировать события в совершенно другом контроллере, причем каждый компонент не знает о другом;
- представления могут содержать другие представления вложенным способом, тем самым улучшая повторное использование;
- изменения в поведении компонента могут быть сделаны без изменения его визуального представления (путем изменения контроллера, а не представления);
- быструю и параллельную разработку;
- тестируемость отдельных компонентов.

Пример приложения

Чтобы упростить сравнение различных фреймворков TypeScript, мы создадим одно и то же приложение с применением каждого из них, что позволит:

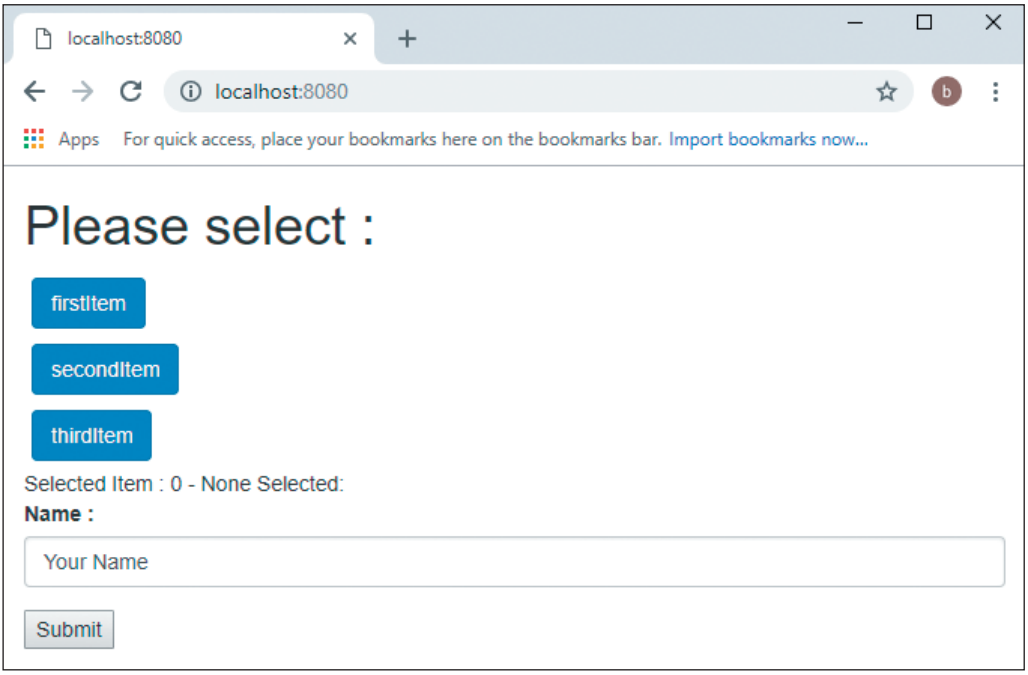
- использовать представление для отображения свойства модели;
- построить массив данных, причем каждый элемент массива представляет собой отдельный экземпляр модели;
- перебрать массив и визуализировать каждый элемент в элементе списка;
- отвечать на событие `click` по каждому визуализированному элементу;
- отобразить свойства модели, которые использовались для визуализации элемента;
- создать форму в нижней части страницы;
- установить значение по умолчанию для элемента управления вводом формы;
- опросить обновленное значение элемента управления формы, когда пользователь отправляет форму.

Мы будем использовать один и тот же набор данных для нашего массива в каждом из четырех примеров приложений. Этот набор данных выглядит так:

```
interface IClickableItem {
   DisplayName: string;
   Id: number;
}

let ClickableItemCollection : IClickableItem[] = ( [
   {Id: 1, DisplayName: "firstItem"},
   {Id: 2, DisplayName: "secondItem"},
   {Id: 3, DisplayName: "thirdItem"},
]);
```

Вначале идет интерфейс с именем `IClickableItem`, у которого есть свойство `Id` типа `number` и свойство `DisplayName` типа `string`. Затем мы определяем массив объектов `IClickableItem` с именем `ClickableItemCollection`. Причина использования такого массива состоит в том, что это очень распространенная структура, с которой можно работать при извлечении данных из конечной точки REST, которая возвращает JSON. Если мы сможем интерпретировать такой простой массив, как этот, то позже мы сможем легко заменить его данными в формате JSON.

Наше итоговое приложение будет выглядеть так:



Здесь мы визуализировали страницу с тремя кнопками, сгенерированными из нашего массива данных. Когда мы нажимаем на любую из этих кнопок, текст `Selected Item` в нижней части страницы будет обновляться. У нас также есть простое поле ввода текста в нижней части страницы, которое помечено как `Name`. По умолчанию это поле ввода будет иметь значение `Your name` при начальной загрузке страницы. При нажатии кнопки **Submit** в нижней части страницы мы запишем значение этого поля ввода в консоль. Задание данных при создании форм и получение этих данных после взаимодействия пользователя с формой подскажет, как каждый из фреймворков обрабатывает привязку данных внутри формы. Хотя эта форма может показаться очень простой, она поможет объяснить различные методы обработки форм, используемые в каждом фреймворке.

Использование Backbone

Мы начнем наше исследование фреймворков TypeScript с создания приложения в Backbone. Хотя можно утверждать, что Backbone не является фреймворком TypeScript как таковым, мы уже видели, как его можно естественным образом использовать с синтаксисом языка TypeScript. Backbone также является одним из старейших фреймворков. Он маленький, легкий и чрезвычайно быстрый.

Backbone, однако, требует написания немного большего количества кода, по сравнению с большинством фреймворков, так как на самом деле это основа MVC-фреймворка. При работе с Backbone вам нужно будет самостоятельно вызывать функции визуализации, а также вручную прикреплять визуализированный HTML-код к дереву DOM.

Чтобы сделать разработку на Backbone немного проще, поверх Backbone был разработан фреймворк Marionette, для упрощения и удаления большей части повторяющегося кода. Фактически существует ряд фреймворков, которые используют Backbone в качестве основного и добавляют дополнительные концепции, которые полезны при создании веб-приложений. Marionette также чрезвычайно быстр, так как добавляет лишь тонкий слой функциональности поверх Backbone, при этом все еще используя базовую библиотеку Backbone.

Производительность визуализации

Если производительность визуализации страниц, циклы ЦП и ОЗУ являются критическими факторами в ваших приложениях, то вы не сможете пройти мимо Backbone ради безупречной скорости визуализации.

В недавнем проекте наша команда участвовала в скоростном тестировании Marionette и Backbone, чтобы определить, подходят ли они для использования на встроенных устройствах. У этих устройств были процессоры с тактовой частотой 400–600 МГц и 128–256 МБ оперативной памяти. По сравнению с современным

настольным компьютером с процессором мощность 3,6 ГГц и 8 ГБ оперативной памяти, это действительно малютки. Это тестовое приложение визуализировало серию из трех экранов HTML, каждый из которых имел различную сложность. Первая страница была очень простой, у второй была переменная структура меню, а третья имела довольно сложное подробное информационное представление ряда данных. Весь код был написан на TypeScript.

Процессоры и оперативная память, доступные на каждом устройстве, были следующими:

- ARM9 с тактовой частотой 400 МГц и 128 МБ ОЗУ;
- PowerPC с тактовой частотой 400 МГц и 256 МБ ОЗУ;
- Marvell PXA300 с тактовой частотой 624 МГц и 256 МБ ОЗУ;
- Core i7 с тактовой частотой 6 ГГц и 8 ГБ ОЗУ (настольный ПК).

Это тестовое приложение запускало таймер высокого разрешения и отправляло сообщения через веб-сокеты на HTML-страницу. Таймер запускался, когда сообщение отправлялось из приложения синхронизации на веб-страницу. Получив сообщение в веб-приложении, он решал, какую страницу отображать. Механизм отображения страницы использовал шаблон Model View Controller для визуализации HTML-кода в браузере. После того как HTML-код был визуализирован на экране, в приложение синхронизации было отправлено сообщение, чтобы остановить часы. Таким образом можно получить информацию о миллисекундной синхронизации, чтобы определить, сколько времени потребовалось для визуализации HTML-кода на каждом из этих устройств.

Этот тест был повторен несколько раз, чтобы получить среднее время визуализации. Результаты приведены в таблице. Все время показано в миллисекундах:

		400 MHz ARM9 128 MB	400 MHz PowerPC 256 MB	624 MHz Marvel 256 MB	3.6 GHz Core i7 8 GB
Backbone	Простая страница	187 мс	131 мс	187 мс	5 мс
	Средняя страница	241 мс	151 мс	267 мс	11 мс
	Комплексная страница	380 мс	370 мс	379 мс	17 мс
Marionette	Простая страница	525 мс	349 мс	501 мс	7 мс
	Средняя страница	1 043 мс	769 мс	851 мс	19 мс
	Комплексная страница	1 532 мс	847 мс	1 014 мс	32 мс

Глядя на результаты в этой таблице, мы видим, что стандартная визуализация Backbone на более медленном процессоре может занять около 130–380 миллисекунд на процессоре PowerPC с тактовой частотой 400 МГц. Однако тот же HTML-код, визуализируемый в Marionette, начинается с 350 миллисекунд и может длиться до 850 миллисекунд. Это различие может быть связано с более высокими

циклами ЦП, которые используются дополнительной обработкой, выполняемой Marionette поверх Backbone.

Таким образом, эти результаты говорят нам, что чем больше логики содержится в фреймворке и, следовательно, чем больше обработки он выполняет, тем медленнее она будет на менее мощных процессорах. Поэтому при оценке структуры, которая делает много магии за кулисами, имейте в виду, что эти свойства будут затрачивать драгоценное время процессора и могут не подходить для более медленных устройств.

Наше решение состояло в том, чтобы использовать Backbone в качестве MVC-фреймворка для этих встроенных устройств, поскольку он предлагал самую высокую скорость из доступных. Тем не менее на современном компьютере с довольно приличным процессором разница между 5 миллисекундами времени визуализации и 32 миллисекундами незначительна.

Настройка Backbone

Настройка среды для Backbone довольно проста и может быть выполнена через tsc и npm следующим образом.

Инициализируйте среду TypeScript с помощью `tsc`:

```
tsc -init
```

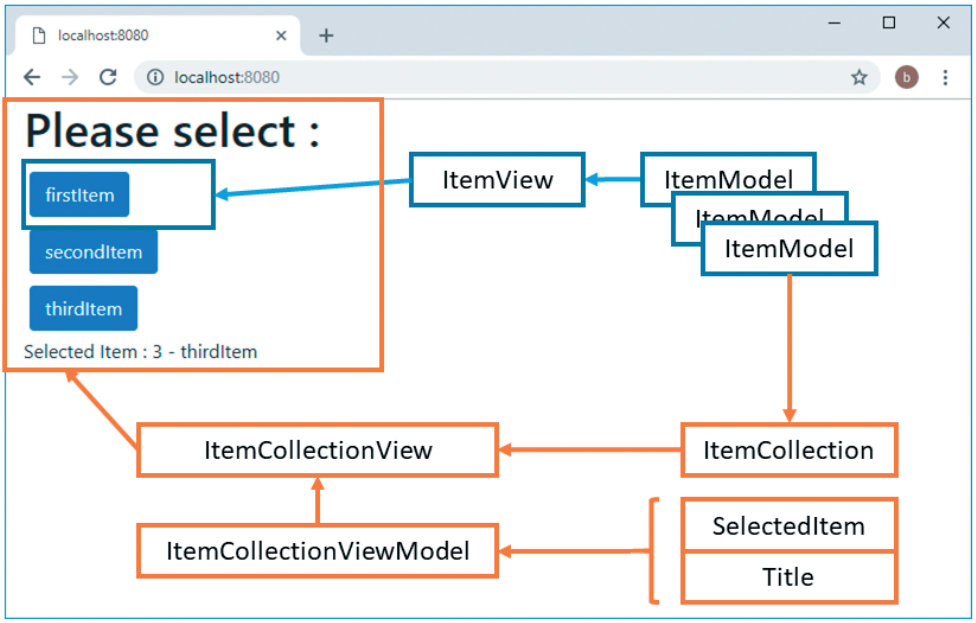
Инициализируйте `npm` и установите Backbone, Bootstrap, JBone и файлы объявлений для Backbone с использованием синтаксиса `@types`:

```
npm init
npm install backbone
npm install bootstrap
npm install jbone
npm install @types/backbone
```

JBone – это реализация jQuery, созданная специально для Backbone. Она включает в себя все функциональные возможности jQuery, которые требуются Backbone, и значительно легче и быстрее, чем полная библиотека jQuery.

Структура Backbone

Первым шагом в создании приложения Backbone является определение элементов, которые нам понадобятся для построения экрана и желаемой функциональности. Как правило, одна модель Backbone будет привязана к одному представлению Backbone. Взаимодействие между моделями и представлениями можно увидеть на следующей диаграмме:



Каждая из наших выбираемых кнопок будет визуализирована в представлении Backbone с именем `ItemView`, которое, в свою очередь, использует одну модель с именем `ItemModel`. Эти модели `ItemModel` затем размещаются в коллекции Backbone с именем `ItemCollection`. Представление `ItemCollectionView` используется для визуализации этой коллекции на экране. У `ItemCollectionView` также есть объект `ItemCollectionViewModel`, который предоставляет два свойства – `SelectedItem` и `Title`. Свойство `Title` визуализирует текст `Please select:` в верхней части экрана, а свойство `SelectedItem` будет использоваться для визуализации текста `Selected Item:` в нижней части экрана, показывая, какой элемент выбран в данный момент.

Модели Backbone

Для нашего приложения мы создадим три модели Backbone. Одна модель – для каждого элемента в нашем массиве `IClickableItem`. Назовем ее моделью `ItemModel`. Вторая модель – для коллекции `ItemModels`, а третья будет использоваться для свойств `SelectedItem` и `Title`. Давайте сначала посмотрим на `ItemModel`:

```
class ItemModel extends Backbone.Model
  implements IClickableItem {
    get Id() {
      return this.get('Id');
    }
```

```
        set Id(value: number) {
          this.set('Id', value);
        }

        get DisplayName() {
          return this.get('DisplayName');
        }

        set DisplayName(value: string) {
          this.set('DisplayName', value);
        }

        constructor(input: IClickableItem) {
          super();
          this.set(input);
        }
    }
```

Здесь у нас есть класс `ItemModel`, который является производным от `Backbone.Model` и реализует интерфейс `IClickableItem`. Мы используем функции получения и установки ES5 для свойств `Id` и `DisplayName`. В рамках этих функций получения и установки мы вызываем функции `model.get ('<propertyName>')` и `model.set ('<propertyName>')`. Это связано с тем, что Backbone хранит свойства объекта внутри себя как атрибуты объекта. Наша функция-конструктор просто вызывает `super`, а затем вызывает Backbone-функцию `set` для установки наших внутренних свойств. Функция `set` используется для гидратации модели Backbone из стандартного объекта или структуры JSON. Это означает, что мы можем создать экземпляр `ItemModel` из объекта JavaScript:

```
let itemModelInstance = new ItemModel({Id: 1,
  DisplayName: 'test'});
```

Способность построить внутренний класс `ItemModel` таким образом означает, что мы можем использовать любой элемент нашего массива `IClickableItem`, и полученная модель Backbone будет корректно гидратироваться. Модели Backbone также предназначены для принятия JSON в качестве входной структуры, что позволяет нам использовать их при обработке результатов запроса конечной точки REST.

Вторая модель, которую мы создадим, используется для размещения всех элементов нашего массива `IClickableItem` и поэтому представляет собой коллекцию экземпляров `ItemModel`:

```
class ItemCollection
  extends Backbone.Collection<ItemModel> {
    model = ItemModel;
  }
```

Этот класс с именем `ItemCollection` наследует от `Backbone.Collection <ItemModel>`, и у него есть единственное свойство `model`. Это свойство установлено в класс `ItemModel`. Поэтому мы можем построить `ItemCollection` следующим образом:

```
let itemCollection = new ItemCollection(ClickableItemCollection);
```

Итак, в одной строке мы создали коллекцию экземпляров `ItemModel` из нашего исходного массива.

Третья модель Backbone, которую мы будем использовать, состоит в размещении свойств `Title` и `SelectedItem`:

```
interface IItemCollectionViewModel {
  Title: string;
  SelectedItem: IClickableItem; }

class ItemCollectionViewModel extends Backbone.Model

implements IItemCollectionViewModel {
  get Title() {
    return this.get('Title');
  }

  set Title(value: string) {
    this.set('Title', value);
  }

  get SelectedItem() {
    return this.get('SelectedItem');
  }

  set SelectedItem(value: IClickableItem) {
    this.set('SelectedItem', value);
  }

  constructor(input?: IItemCollectionViewModel) {
    super();
    this.set(input);
  }
}
```

Здесь мы определили интерфейс с именем `IItemCollectionViewModel`, у которого есть свойство `Title` типа `string` и свойство `SelectedItem` типа `IClickableItem`. Свойство `SelectedItem` будет использоваться для хранения элемента, выбранного пользователем. Затем мы определяем класс с именем `ItemCollectionViewModel`, который реализует этот интерфейс, и у него должно быть два свойства – `Title` и `SelectedItem`. Как мы видели при работе с предыдущими моделями Backbone, мы используем функции получения и установки

ES5 для вызова внутренних функций `get` и `set`, чтобы сохранить эти свойства в атрибутах внутренних объектов Backbone. Наш конструктор используется для гидратации модели из стандартного объекта.

Класс ItemView

Как упоминалось ранее, нам нужно будет построить два представления. Первое представление будет называться `ItemView` и использоваться для визуализации одной кнопки в DOM. Как таковое оно привязано к `ItemModel` в нашей коллекции. Давайте посмотрим на этот класс `ItemView`:

```
class ItemView extends Backbone.View<ItemModel> {
   template: (json: any, options?: any) => string;
   constructor(
      options = <Backbone.ViewOptions<ItemModel>>{}
   ) {
      options.events = { 'click': 'onClicked' };
      super(options);
      let templateSnippet = $('#itemViewTemplate').html();
      this.template = _.template(templateSnippet);
      _.bindAll(this, 'onClicked');
   }
   render() {
      this.$el.html(
         this.template(this.model.attributes)
      );
      return this;
   }
   onClicked() {
      EventBus.Bus.trigger("item_clicked", this.model.attributes);
   }
}
```

Наш класс `ItemView` наследует от `Backbone.View` и использует синтаксис обобщений для строгой типизации модели, применяемой этим представлением, в классе `ItemModel`, который мы создали ранее. Функция-конструктор принимает необязательный аргумент с именем `options` типа `Backbone.ViewOptions <ItemModel>`, который по умолчанию также установлен как пустой объект `{}`. Мы можем обратиться к файлу определения Backbone, чтобы увидеть, какие опции можно отправить через это свойство.

Обратите внимание, что мы определили для этого класса свойство с именем `template` и определили тип этого свойства как функцию, которая принимает два свойства и возвращает строку. Это необходимо для установки свойства представления `template`, поскольку в стандартном файле определения для `Backbone.View` свойство `template` закомментировано.

Наша функция-конструктор начинается с установки свойства `events` для объекта `options`. Свойство `events` используется представлением, чтобы реагировать на определенные события DOM, такие как `keydown`, `keyup` или `click`. Каждое из этих событий будет вызывать функцию в представлении. В данном случае это функция `onClicked`. После настройки данных параметров наш конструктор передает объект `options` в базовый класс `Backbone.View` через вызов super.

Затем конструктор устанавливает свойство класса `template`, в котором находится HTML-код, который этот класс будет использовать для визуализации на экране. Обратите внимание, что мы используем синтаксис jQuery для извлечения HTML-кода элемента DOM с идентификатором `itemViewTemplate`, который будет использоваться в качестве нашего HTML-шаблона. Поэтому наша HTML-страница должна будет содержать следующий сценарий, чтобы Backbone мог подобрать этот шаблон:

```
<script type="text/template" id="itemViewTemplate">
   <button style="margin: 5px;" class="btn btn-primary">
      <%= DisplayName %>
   </button>
   <br/>
</script>
```

Мы определили тег `<script>` в нашем HTML-коде типа `"text/template"` с идентификатором `itemViewTemplate`. В этом сценарии мы определяем элемент `<button>` со свойствами `style` и `class`. Обратите внимание, что Backbone по умолчанию использует шаблонизатор Underscore. Для визуализации свойства модели в HTML мы используем в шаблоне синтаксис `<% = PropertyName%>`. Это означает, что `<% = DisplayName%>` заменит свойство `DisplayName` из нашей модели на результирующий HTML-код.

Затем наш класс `ItemView` определяет функцию `render`. Эта функция вызывает функцию html внутреннего свойства `$el` и передает результат вызова функции `template` со свойством внутренней модели `attribute`. Это свойство на самом деле – простой объект со всеми свойствами внутренней модели. Данный вызов возьмет модель Backbone, объединит ее с HTML-шаблоном, сгенерирует результирующий HTML-код, а затем обновит DOM. Обратите внимание, что он также возвращает `this`, поэтому любой код, который использует его, может связывать команды в функцию `render`.

Последняя функция класса `ItemView` – `onClicked`, которая будет вызываться при нажатии кнопки, визуализированной в DOM. Backbone предоставляет очень простую шину сообщений, которую можно использовать для классов, чтобы уведомлять друг друга о конкретных событиях. Чтобы использовать эту шину событий, мы создадим очень простой класс TypeScript со статической функцией, с помощью которой мы либо запускаем события, либо прослушиваем их:

```
class EventBus {
   static Bus = _.extend({}, Backbone.Events);
}
```

У класса `EventBus` есть единственное статическое свойство `Bus`, которое использует функцию `extends` с подчеркиванием для объединения пустого объекта JavaScript `{}` с объектом `Backbone.Events`. Это все, что требуется для включения полноценной шины событий в наше приложение Backbone.

Запуск соответствующего события из функции `onClicked` тогда просто превращается в:

```
onClicked() {
   EventBus.Bus.trigger("item_clicked", this.model.attributes);
}
```

Здесь мы запускаем событие `item_clicked` и присоединяем свойство модели `attributes`, которое использовалось в `ItemView`, в качестве параметра к функции `trigger`.

Класс ItemCollectionView

Второе представление, которое мы создадим, – это представление для визуализации свойств Title и SelectedItem и коллекции ItemView:

```
class ItemCollectionView extends Backbone.
View<ItemCollectionViewModel> {
   template: (json: any, options?: any) => string;
   itemCollection: ItemCollection;
   constructor(
      options: Backbone.ViewOptions<ItemCollectionViewModel> = {},
      _itemCollection: ItemCollection) {
      super(options);
      this.itemCollection = _itemCollection;
      let templateSnippet = $(`#itemCollectionViewtemplate`).html();
      this.template = _.template(templateSnippet);
      this.listenTo(EventBus.Bus,"item_clicked", this.handleEvent);
   }
   render() {
      this.$el.html(this.template(this.model.attributes));
      this.itemCollection.each((item) => {
         var itemView = new ItemView({ model: item });
         this.$el.find('#ulRegions').append(itemView.render().el);
      });
      return this;
   }
   handleEvent(e: ItemModel) {
```

```
        this.model.SelectedItem = e;
        this.render();
    }
}
```

Здесь мы определили класс с именем `ItemCollectionView`, который использует синтаксис обобщений для строгой типизации модели Backbone в тип `ItemCollectionViewModel`. Мы определили наше стандартное свойство template, а также свойство `itemCollection`, которое будет использоваться для хранения коллекции `IClickableItems`. У нашей функции-конструктора есть два параметра, `options` и `_itemCollection`. Параметр `options` – это стандартный тип `Backbone.ViewOptions`, а параметр `_itemCollection` имеет тип `ItemCollection`.

Конструктор инициализирует базовый класс вызовом `super`, а затем устанавливает внутреннее свойство `itemCollection`. Шаблон, который будет использоваться для визуализации этого представления, затем настраивается путем извлечения HTML-кода из тега сценария с идентификатором `itemCollectionViewTemplate`. Этот шаблон определен на нашей HTML-странице:

```
<script type="text/template" id="itemCollectionViewtemplate">
    <h1> <%= Title %> </h1>
    <div id="ulRegions">
    </div>
    <div> Selected Item :
        <%= SelectedItem.Id %> -
        <%= SelectedItem.DisplayName %>
    <div>
</script>
```

В этом шаблоне есть теги `<h1>`, в которые будет заключено свойство `Title`. Далее идет тег `<div>` с идентификатором `ulRegions`. Тут кнопки будут внедрены в DOM, когда мы будем визуализировать это представление. Подробнее об этом чуть позже. Заключительная часть данного шаблона визуализирует атрибуты `Id` и `DisplayName` свойства `SelectedItem`, чтобы мы могли видеть, какая кнопка была выбрана.

Если мы теперь посмотрим на функцию `render` в этом представлении, то заметим, что она делает немного больше, чем в предыдущем представлении. После вызова функции `html` для свойства `$el` с `template` и `model` DOM фактически получит полностью визуализированный шаблон, дополненный атрибутами модели. Затем функция `render` проходит через каждый из элементов в `itemCollection` и создает для каждого из них `ItemView`. После этого она добавляет результат `render().el` для этого `ItemView` к элементу DOM с именем `ulRegions`. Таким образом, по сути, функция `render` в этом представлении вызывает функцию `render` для каждого `ItemView` и объединяет результаты.

Единственное, что осталось рассмотреть в этом представлении, – это последняя строка функции конструктора, где мы вызываем функцию `listenTo` с тремя аргументами. Первый аргумент – это шина событий, как было определено ранее, второй – событие, которое нужно прослушивать, а третий аргумент – это функция, вызываемая при возникновении этого события. Это означает, что при щелчке по отдельному `ItemView` и запуске события `item_clicked` данное представление будет перехватывать событие и вызывать функцию `handleEvent`.

Функция `handleEvent` просто устанавливает внутреннее свойство модели `SelectedItem` в значение входящего события с именем `e`. Помните, что наш `ItemView` будет вызывать событие `item_clicked` с атрибутами своей внутренней модели. Это означает, что аргумент `e` будет содержать свойства `Id` и `DisplayName` и поэтому может быть строго типизирован в `ItemModel`. В конце функция `handleEvent` вызывает функцию `render`.

Приложение Backbone

Создав две модели, коллекцию моделей и два представления, мы теперь создадим контроллер для связывания этих элементов. В Backbone, однако, нет специального класса контроллера, но мы можем использовать стандартный класс TypeScript для выполнения данной работы. Этот класс будет называться ScreenViewApp:

```
class ScreenViewApp {
   start() {
      let itemCollection = new ItemCollection(ClickableItems);
      let collectionItemViewModel = new ItemCollectionViewModel({
         Title: "Please select :",
         SelectedItem: {
            Id: 0, DisplayName: 'None Selected:'
         }
      });
      let itemCollectionView = new ItemCollectionView({
         model: collectionItemViewModel
         }, itemCollection
      );
      $(`#pageLayoutRegion`).html(
         itemCollectionView.render().el
      );
   }
}
```

Мы создали класс `ScreenViewApp` с единственной функцией `start`. Тело функции показывает, как мы создаем различные элементы, которые используются для визуализации нашего приложения на экране.

Сначала функция `start` создает новую модель `ItemCollection` из массива `ClickableItems`, после чего устанавливает модель `ItemCollectionViewModel` со свойствами `Title` и `SelectedItem` с начальными значениями по умолчанию. Затем мы создаем экземпляр класса `ItemCollectionView` с необходимыми аргументами. Обратите внимание, что первый аргумент на самом деле является объектом с единственным свойством `model`, которое установлено как экземпляр нашего объекта `collectionViewModel`. Это соглашение Backbone, где мы можем создать представление Backbone с рядом различных свойств в конструкторе, включая события, `tagNames`, `classNames` и др. Опять же, обратитесь к документации или файлу определения для получения информации о дополнительных опциях и их использовании.

Последняя строка функции `start` просто вызывает функцию `html` для DOM-элемента `pageLayoutRegion` для установки визуализированного HTML-кода. Обратите внимание, что после визуализации представления Backbone окончательный HTML-код помещается в свойство `el` самого представления.

Разобравшись с этим классом, мы теперь можем вызвать функцию `start` с нашей страницы `index.html`:

```
<script >
   window.onload = function() {
      app = new ScreenViewApp();
      app.start();
   }
</script>

<div id="pageLayoutRegion">
</div>
```

Здесь мы создаем экземпляр класса `ScreenViewApp` в рамках функции `window.onload`, а затем вызываем функцию `start`. Под закрывающим тегом `<script>` мы поместили теги `<div>` с идентификатором `pageLayoutRegion`, куда будет помещен результирующий HTML-код.

Формы

Теперь, когда у нас есть базовое приложение Backbone, мы можем выполнить несколько настроек, чтобы вставить форму в нижнюю часть страницы и показать, как получать и устанавливать значения формы в рамках нашего представления. Эта форма будет обрабатываться классом `ItemCollectionView`, поэтому нам нужно обновить нашу модель новым свойством:

```
class ItemCollectionViewModel extends Backbone.Model
implements IItemCollectionViewModel {
   ...существующие свойства
```

```
    set Name(value: string) {
      this.set('Name', value);
    }

    get Name() {
      return this.get('name');
    }

    ...существующий конструктор

  }
```

Здесь мы просто добавили соответствующую пару функций ES5 `get` и `set` для хранения и извлечения свойства `Name`. Как только это изменение будет выполнено, нам также нужно будет установить это свойство при создании самой `ItemCollectionViewModel`. Этот код находится в файле `app.ts`, в функции `start`:

```
  class ScreenViewApp {
    start() {

      ...существующий код

      let collectionItemViewModel = new ItemCollectionViewModel({
        Title: "Please select :",
        SelectedItem: {
          Id: 0, DisplayName: 'None Selected:'
        },
        Name: "Your Name"
      });
      ...существующий код
    }

  }
```

Здесь мы обновили функцию start в классе `ScreenViewApp`, добавив свойство `Name` при создании `ItemCollectionViewModel`. Это свойство будет по умолчанию иметь значение `"Your Name"`.

Следующей модификацией нашего кода будет сам шаблон в файле `index.html`:

```
  <script type="text/template" id="itemCollectionViewtemplate">
    ...существующий HTML-шаблон

    <div class="form-group">
      <label for="inputName">Name :</label>
      <input type="text" class="form-control input-name"
```

```
        id="inputName" value="<%= Name %>" />
    </div>
    <button id="submit-button-button"
      class="submit-button">Submit</button>
</script>
```

Здесь мы обновили HTML-шаблон `itemCollectionViewTemplate`, добавив теги `<div>` со стилем `form-group` из Bootstrap. В эти теги мы добавили теги `<label>` и `<input>` с типом `"text"`. Обратите внимание, что мы также добавили атрибут `id = "inputName"`, чтобы иметь возможность ссылаться на этот элемент ввода по `id`. В конце мы добавили теги `<button>` с идентификатором `"submit-button-button"` и классом `"submit-button"` из Bootstrap.

После внесения изменений в шаблон нам нужно будет отследить событие, когда пользователь нажимает кнопку отправки, чтобы мы могли запросить значение, введенное пользователем в форму. Это очень похоже на способ, которым мы присоединили функцию к событию `click` для каждой из наших кнопок. Мы обновим класс `ItemCollectionView`:

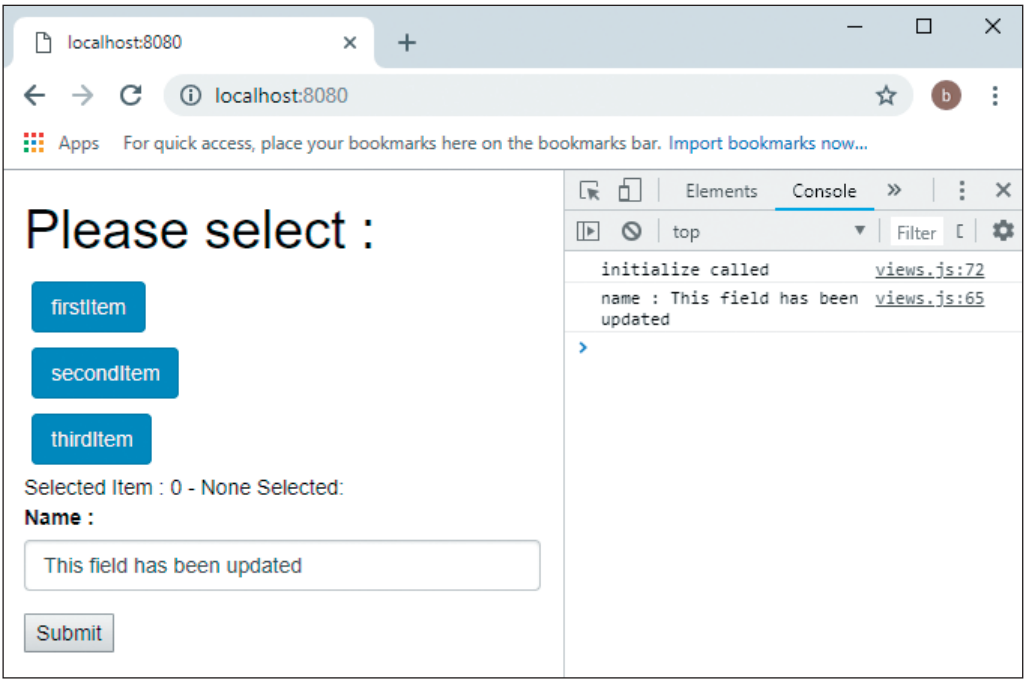
```
class ItemCollectionView extends Backbone.
View<ItemCollectionViewModel> {
    template: (json: any, options?: any) => string;
    itemCollection: ItemCollection;
    constructor(
      options: Backbone.ViewOptions<ItemCollectionViewModel> = {},
      _itemCollection: ItemCollection)
      {
        options.events = {'click #submit-button-button':
          'submitClick'}
        super(options);
        ...существующий код
      }
    submitClick() {
      let name = this.$el.find('#inputName');
      if (name.length > 0) {
        console.log(`name : ${name.val()}`);
      } else {
        console.log(`cannot find #inputName`);
      }
    }
}
```

Мы внесли в класс `ItemCollectionView` два изменения. Во-первых, мы добавили свойство `events` к аргументу `options` в первой строке функции-конструктора. Это очень похоже на то, как мы захватили событие `click` в `ItemView` ранее. Здесь, однако, мы определили, что интересующее нас событие `click` находит-

ся в элементе с идентификатором `submit-button-button`. Backbone использует стандартный синтаксис JQuery, применяя `#submit-button-button`. Когда данное событие запускается, мы будем обрабатывать это с помощью функции `submitClick`.

Функция `submitClick` начинается с поиска элемента, который присоединяется к представлению, вызывая функцию `this.$el.find`. Помните, что мы можем обращаться к DOM в нашем представлении, используя свойство `$el`. Функция `find` также применяет синтаксис JQuery для поиска элемента DOM с идентификатором `inputName`, который фактически является нашим полем ввода. Если мы находим элемент с этим идентификатором, то извлекаем введенное пользователем значение данного поля, вызывая функцию `val()`.

На приведенном ниже скриншоте показан результат обновления формы и нажатия кнопки `Submit`:



Это основы работы с данными формы в Backbone. Как мы уже видели, мы можем использовать значения модели Backbone в своих HTML-шаблонах, как и любое другое свойство модели. Мы также можем реагировать на события нажатия на конкретном элементе DOM, обращаясь к нему по свойству `id`. Когда форма отправлена, мы можем извлечь значения элемента формы, запросив его значение с помощью JQuery.

Резюмируя

Как мы отмечали в начале этого раздела, Backbone требует, чтобы мы писали немного больше кода, чем другие платформы, дабы завершить свое приложение. Тем не менее, с другой стороны, сам код очень читабелен и логичен и не содержит никакой магии. Поэтому очень просто взять и написать его, и не нужно много времени, чтобы научиться использовать его. Одним из величайших преимуществ фреймворка без излишеств, такого как Backbone, является его абсолютная скорость.

Использование Aurelia

Aurelia был одним из первых фреймворков SPA, который предложил полную интеграцию TypeScript. Это фреймворк, который специально использует возможности ECMAScript 6 для улучшения опыта разработки. Одной из самых поразительных особенностей Aurelia является небольшое количество кода, которое вам нужно написать, чтобы добиться цели. Aurelia понимает, что если вы пишете стандартный класс, то, вероятно, захотите использовать свойства класса для визуализации HTML-кода. Из всех фреймворков, которые мы будем обсуждать, Aurelia – самый простой в использовании и самый интуитивно понятный. В нем нет скрытых ошибок или специальных обходных путей. Этот фреймворк пошел на многое, чтобы упростить процесс разработки на TypeScript.

Настройка Aurelia

Самый простой способ настроить среду разработки – использовать интерфейс командной строки Aurelia `aurelia-cli`, который можно установить следующим образом:

```
npm install aurelia-cli -g
```

После установки его можно вызвать для создания нового проекта:

```
au new
```

Вам будет предложен простой набор вопросов, начиная с имени базового каталога, который вы хотели бы использовать. Следующий вопрос – использовать ли ESNext или TypeScript в качестве языка разработки, и последний вопрос – загружать или нет все зависимости проекта. Выберите TypeScript и затем Yes, чтобы загрузить зависимости. Потребуется несколько минут, чтобы настроить структуру проекта по умолчанию. После этого будет создан новый каталог на основе имени проекта, выбранного вами в начале процесса.

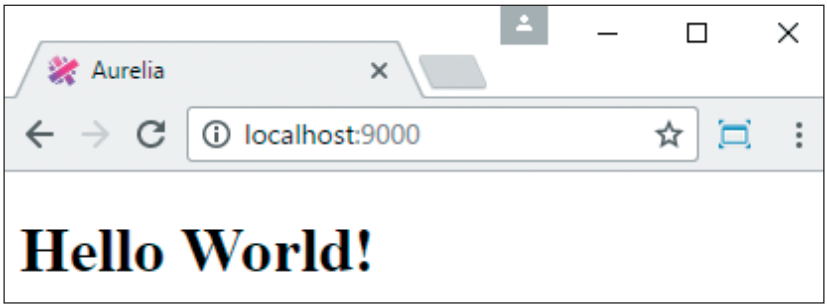
У программы aurelia-cli есть несколько вариантов. Чтобы скомпилировать свой проект, введите:

```
au build
```

Чтобы запустить только что созданное приложение Aurelia, введите:

```
au run
```

Далее последуют шаги компиляции и связывания, а затем настройка http-сервера для обслуживания приложения по умолчанию на порту `9000`. При переходе по ссылке `http://localhost:9000` отобразится экран Aurelia по умолчанию:



Контроллеры и модели Aurelia

Aurelia объединяет контроллеры и модели в единую классовую структуру. Это означает, что стандартный класс TypeScript действует как контроллер, а его свойства действуют как его модель. Все классы Aurelia построены на стандарте ECMAScript 2016. Это означает, что тут нет методов получения и установки, которые были необходимы нам в Backbone, и не требуется никаких специальных конструкторов для гидратации модели. Классы могут содержать другие классы, что позволяет очень легко определять сложные и вложенные модели. Давайте изменим файл `src/app.ts`:

```
interface IClickableItem {
   DisplayName: string;
   Id: number;
}

export class App {
   Title = 'Please select :';
   SelectedItem: IClickableItem =
      {Id: 0, DisplayName: "None selected”};
   items: IClickableItem[] = ([
      {Id: 1, DisplayName: “firstItem”},
      {Id: 2, DisplayName: “secondItem”},
      {Id: 3, DisplayName: “thirdItem”},
   ]);
}
```

Вначале идет стандартное определение интерфейса `IClickableItem`. Затем мы изменили класс `App` и добавили три свойства. Первое свойство `Title` будет содержать текст в верхней части экрана. Второе свойство `SelectableItem` будет содержать наш текущий выбранный элемент. Последнее свойство `items` содержит наш массив `IClickableItem`.

И это все, что нужно сделать.

В Aurelia стандартные классы используются в качестве контроллера, а стандартные свойства классов – в качестве моделей.

Представления Aurelia

Aurelia использует соглашение об именовании, чтобы привязывать классы (или контроллеры) к их представлениям. Наш класс называется `App`, поэтому среда выполнения Aurelia будет искать в том же каталоге, что и класс, чтобы обнаружить HTML-шаблон с таким же именем. Поэтому он будет привязывать `app.ts` к `app.html` и использовать `app.html` в качестве шаблона.

Мы изменим существующий файл `app.html` с помощью следующего фрагмента:

```
<template>
   <require from=”bootstrap/dist/css/bootstrap.css”></require>
   <h1>${Title}</h1>
   <ul>
      <div repeat.for=”item of items” >
         <button style=”margin: 5px;” class=”btn btn-primary”>
            ${item.DisplayName}</button>
      </div>
   </ul>
   <div>
      Selected Item : ${SelectedItem.Id} -
         ${SelectedItem.DisplayName}
   </div>
</template>
```

Здесь мы заключили фрагмент HTML-кода в теги `<template>`. В первой строке этого шаблона используется тег `<require>` со свойством `from`, которое обращается к файлу `bootstrap.css`. Это соглашение Aurelia для включения внешних зависимостей в шаблон. Обратите внимание, что нам нужно установить Bootstrap, используя npm, как обычно, прежде чем этот файл станет доступным. Затем шаблон определяет тег `<h1>`, который использует синтаксис `${propertyName}` для визуализации свойства модели с именем `Title`. Далее идут теги `<ul>`, а внутри них теги `<div>`.

Тег `<div>` в данном случае является интересной частью этого шаблона. Обратите внимание, что мы ввели атрибут с именем `repeat.for` = “`item of items`”. Этот

синтаксис специфичен для Aurelia и указывает на то, что будет выполняться проход через свойство `items` класса `App` и тег `<div>` будет повторяться для каждого отдельного элемента. Затем данный синтаксис будет доступен для тега `<div>` через переменную item. Мы могли бы вызвать `arrayItem`, и в этом случае в коде появилось бы это: `repeat.for` = “`arrayItem of items`”.

Внутри тега `<div>` мы просто создаем тег `<button>` и используем синтаксис привязки данных `$ {item.DisplayName}` для визуализации свойства `DisplayName` каждого элемента массива.

Начальная загрузка приложения

Имея представление и модели, мы можем обратить наше внимание на файл `index.ejs` в корне каталога проекта. Он выглядит так:

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset=”utf-8”>
      <title><%- htmlWebpackPlugin.options.metadata.title %>
         </title>
      <meta name=”viewport” content=”width=device-width,
         initial-scale=1”>
      <base href = “<%- htmlWebpackPlugin.options.metadata.
         aseUrl %>”>
      <!-- И export class App импортированные CSS-стили
         конкатенируются и добавляются автоматически -->
   </head>
   <body aurelia-app=”main”>
      <% if (htmlWebpackPlugin.options.metadata.server) { %>
      <!-- Перезагрузка Webpack Dev Server -->
      <script src=”/webpack-dev-server.js”></script>
      <% } %>
   </body>
</html>
```

Это короткий, но довольно сложный HTML-файл, в котором используется библиотека шаблонов EJS для внедрения значений в HTML-файл и выполнения `if-` операторов. Мы не будем изменять этот файл для нашего приложения, а просто отметим тег `<body>`.

У тега `<body>` есть дополнительный атрибут `aurelia-app`, который имеет значение `main`. Этот атрибут сообщает Aurelia, что он должен искать файл `main.js` в каталоге `src` в качестве начальной отправной точки для приложения. Если мы теперь посмотрим на файл `main.js` в каталоге `src`, мы найдем экспортированную функцию `configure`, а в нижней части функции – интересную строку:

```
return aurelia.start().then(
   () => aurelia.setRoot(PLATFORM.moduleName('app'))
);
```

Эта строка вызывает функцию `start` для объекта `aurelia`, а после выполнения этого промиса вызывает функцию `setRoot` с аргументом `PLATFORM.moduleName('app')`. Эта строка фактически дает Aurelia сигнал, что он должен загрузить модуль `app` в качестве точки входа в приложение.

С этими драгоценными строками кода у нас есть работающее приложение.

События

Следующее требование для нашего приложения – визуализация выбранного в данный момент элемента, когда пользователь нажимает одну из наших кнопок. Для этого нам нужно добавить функцию в класс `App`, которая будет действовать как обработчик события, а затем привязать DOM-событие `onclick` к этой функции. Давайте сначала изменим наш шаблон `app.html`:

```
<div repeat.for="item of items"
   click.delegate="onItemClicked(item)">
   <button > ...
</div>
```

Мы добавили в тег `<div>` атрибут `click.delegate` и внутри него вызываем функцию `onItemClicked` с `item` в качестве аргумента. Обратите внимание, что этот атрибут определяется не внутри тега `<div>`, а на уровне самого тега. Это означает, что функция `onItemClicked` должна быть присоединена к классу `App`, а не к классу `ClickableItem`, что несколько отличается от реализации Backbone, где каждый отдельный `ItemView` получает событие `onclick`.

После этого мы можем изменить класс `App` следующим образом:

```
export class App {
   Title: string = 'Please select:';
   ... Существующий код;
   onItemClicked(event: IClickableItem) {
      this.SelectedItem = event;
   }
}
```

Здесь мы добавили функцию `onItemClicked` с единственным аргументом `event` типа `ClickableItem`. В этой функции мы просто присваиваем данный аргумент свойству `SelectedItem`. Aurelia автоматически обнаружит, что свойство `SelectedItem`, которое используется в шаблоне `app.html`, изменилось и повторно визуализирует представление.

Формы

Как мы видели в большинстве случаев при работе с Aurelia, добавлять формы в HTML-код и сохранять их значения в модели очень просто. Давайте обновим наше представление Aurelia, добавив свойство Name, которое установит значение по умолчанию для элемента управления формы. Затем мы извлечем значение этого свойства в событии click. Сперва мы обновим класс App:

```
export class App {
   ...существующие свойства
   Name = 'Your Name';
   ...существующий код
   onSubmitClicked() {
      console.log(`onSubmitClicked : Name : ${this.Name}`)
   }
}
```

Здесь мы просто добавили свойство Name в наш класс и установили его значение по умолчанию как 'Your Name'. Мы также создали функцию onSubmitClicked, которая записывает значение этого свойства в консоль.

Теперь мы можем включить свойство Name в форму, обновив шаблон app.html:

```
<template>
   ...существующий шаблон

   <  div class="form-group">
      <label for="inputName">Name :</label>
      <input type="text" class="form-control input-name"
         id="inputName" value.bind="Name" />
   </div>

   <button id="submit-button-button" class="submit-button"
      click.delegate="onSubmitClicked()">Submit</button>
</template>
```

Здесь мы добавили тег в шаблон, в котором есть класс formgroup из Booststrap, и создали тег <label> и элемент <input>, аналогично тому, как мы делали, работая с Backbone. Интересной особенностью элемента <input> является новый атрибут value.bind, для которого установлено значение Name. Так Aurelia понимает, что должен привязать значение свойства Name к этому элементу управления формы. Процесс связывания элемента управления со свойством означает, что изменения в одном будут автоматически отражаться в другом. Поэтому, когда мы установим значение этого свойства в классе, оно обновит элемент управления формы, и, аналогично, когда пользователь обновляет элемент управления формы на странице, значение свойства будет автоматически обновляться.

Резюмируя

Как мы убедились, построение MVC-кода в Aurelia – очень простое занятие. Используя возможности классов ECMAScript 2016 и простые HTML-шаблоны, мы можем выполнить большую работу с помощью очень небольшого количества кода. Aurelia также понимает, что когда мы создаем модели для визуализации информации пользователю, у нас может возникнуть желание обновить эти модели на основе пользовательского ввода. Синтаксис, который Aurelia использует для привязки моделей к пользовательскому вводу, действительно очень прост.

Angular

Как упоминалось в начале этой главы, Angular 2 – это полностью переписанный фреймворк Angular 1, который использовал TypeScript в качестве предпочтительного языка. Соглашение об именовании, принятое командой Angular, гласит, что Angular 1 теперь носит название AngularJs, а Angular версии 2 и выше называется Angular. Со времени выхода Angular 2 команда Angular выпустила ряд крупных обновлений, и текущая версия Angular на момент написания этой главы – Angular 7. Примеры, приведенные в этой книге, написаны с использованием Angular 7, поэтому везде, где вы видите слово Angular, помните, что оно относится к Angular версии 7. В этом разделе мы рассмотрим, как шаблон проектирования Model View Controller используется в Angular.

Установка Angular

Подобно настройке среды разработки в Aurelia, в Angular также есть инструмент для настройки проекта с помощью командной строки под названием «Интерфейс командной строки Angular». Его можно установить с помощью `npm`:

```
npm install -g @angular/cli
```

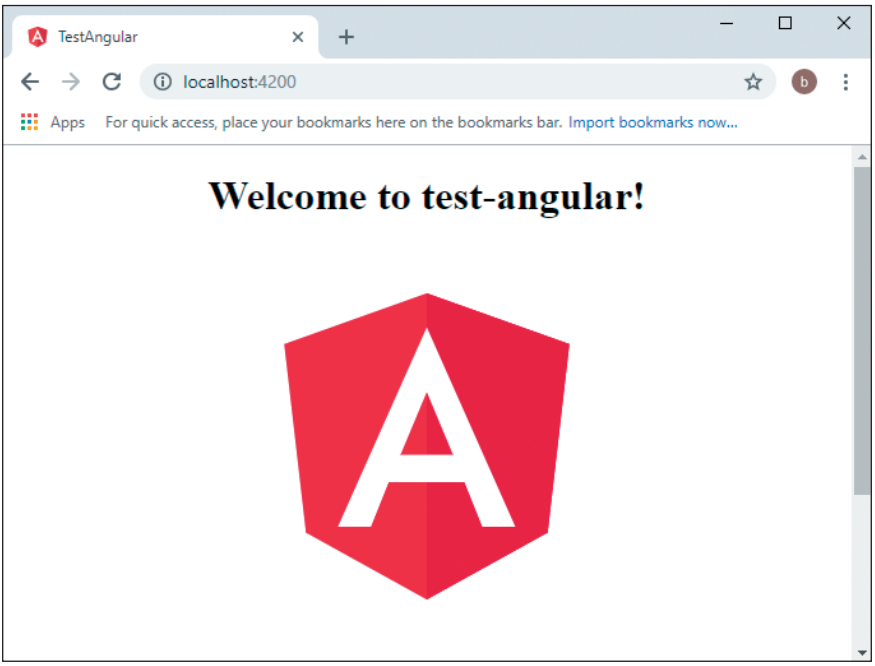
После того как интерфейс командной строки будет установлен глобально, мы можем настроить среду разработки Angular с помощью интерфейса командной строки:

```
ng new my-app
```

Интерфейс командной строки Angular обозначается как ng, и здесь мы указали, что он должен создать новый проект в новом каталоге с именем `my-app`. В новом каталоге интерфейс командной строки загрузит и установит все необходимые компоненты приложения Angular, а также создаст минимальный пример проекта в каталоге `src/app`, чтобы было с чего начать. Чтобы запустить веб-сервер разработки и увидеть, как работает это приложение, введите следующую команду:

```
npm start
```

Команда start скомпилирует весь исходный код приложения и запустит веб-сервер на порту `4200`. Перейдя по ссылке `http://localhost:4200`, вы увидите, что содержит это приложение, как показано на приведенном ниже скриншоте:



Наряду с компиляцией приложения и автоматическим запуском веб-сервера команда `npm start` также будет просматривать исходные файлы в каталоге проекта и автоматически перекомпилировать приложение после изменения файлов. Она также будет давать веб-браузеру сигнал перезагрузить приложение. Встроенные возможности наблюдения, перекомпиляции и перезагрузки очень помогают в разработке веб-приложений, обеспечивая быструю обратную связь при изменении исходного кода.

Следите за консолью, на которой вы запускаете `npm start`. Она покажет все ошибки компиляции TypeScript, возникающие при сохранении ваших файлов.

Модели Angular

Модели и контроллеры Angular такие же, как и модели Aurelia, в том смысле, что они являются простыми классами. Давайте начнем с редактирования файла `app/app.component.ts` и добавим свои модели Angular и контроллер:

```
export class ClickableItem {
   displayName: string;
   id: number;
}

let ClickableItemArray: ClickableItem[] = [
   {id: 1, displayName: "firstItem"},
   {id: 2, displayName: "secondItem"},
   {id: 3, displayName: "thirdItem"},
];

// Существующий код @Component;

export class AppComponent {
   Title = 'Please select :';
   items = ClickableItemArray;
   SelectedItem: ClickableItem =
      {Id: 0, DisplayName: "None selected"};
}
```

Здесь у нас есть стандартный класс `ClickableItem`, у которого есть свойства `DisplayName` и `Id`. Затем мы создаем массив `ClickableItemArray` для хранения элементов нашего массива, как мы видели ранее. Наш последний класс называется `AppComponent`, и у него есть свойства `Title` и `items`, как и у класса модели для Aurelia. У нас также есть свойство `SelectedItem` для хранения значения текущего выбранного элемента. Обратите внимание, что мы устанавливаем значение по умолчанию свойства `SelectedItem` равным `0`, `None Selected`. Это обеспечит правильную инициализацию данного свойства при создании класса.

Представления Angular

Angular использует декоратор класса с именем `@Component`, чтобы указать, что класс может выступать в качестве HTML-компонента. Давайте внимательнее посмотрим на этот декоратор:

```
import { Component } from '@angular/core';
... Существующий код ClickableItem;

@Component({
   selector: 'app-root',
   templateUrl: './app.component.html',
   styleUrls: ['./app.component.css']
})
export class AppComponent {
   ...существующий код
}
```

Этот код начинается с оператора `import`, чтобы определить декоратор компонента из библиотеки `'@angular/core'`. Оператор `import` является частью модульного синтаксиса, который мы рассмотрим в следующей главе, и дает нам возможность с легкостью обращаться к другим классам из фреймворка Angular. Модуль `Component`, который мы импортируем, фактически является функцией декоратора класса, которая будет вставлять свойства Angular в наш стандартный класс TypeScript.

Декоратор `@Component` определяет три свойства: `selector`, `templateUrl` и `styleUrls`. Свойство `selector` используется для обращения к элементу DOM HTML, где будет визуалировано представление, аналогично стандартному селектору jQuery. Свойство `templateUrl` указывает, где находится HTML-файл шаблона для этого компонента, а свойство `styleUrls` используется для включения CSS-файлов, используемых компонентом. В тех случаях, где Aurelia сопоставляла HTML-файл шаблона по ассоциации имен, Angular использует свойство класса `templateUrl`.

Давайте посмотрим на код этого шаблона в файле `app.component.html`:

```
<h1>
   {{Title}}
</h1>
<ul>
   <div *ngFor="let item of items">
      <button class="btn btn-primary">{{item.DisplayName}}</button>
      <br />
   </div>

</ul>

<div *ngIf="SelectedItem">
   <div>Selected : {{SelectedItem.Id}}
      - {{SelectedItem.DisplayName}}
   </div>
</div>
```

В этом шаблоне мы используем синтаксис Angular `{{propertyName}}` для обращения к свойствам внутри нашей модели. Он напоминает синтаксис `${propertyName}`, используемый в Aurelia. В нашем шаблоне, также похожем на шаблон Aurelia, есть элемент `<h1>` для отображения свойства `Title`, и обычные теги `<ul>` и дочерние теги `<div>`.

Внутри тега `<div>` мы снова перебираем свойство `items` в модели `AppComponent`, чтобы визуализировать каждый элемент массива в DOM. Angular использует ключевое слово `* ngFor` в своем шаблоне для обозначения конструкции цикла. `"let item of items"` снова обращается к каждому элементу массива в шаблоне через имя переменной `item`. Как мы видели, когда работали с Aurelia, если мы изменим эту фразу на `"let arrayItem of items"`, то нам нужно будет обращаться к каждому элементу массива через переменную `arrayItem`.

Шаблон, который будет использоваться для каждого элемента массива, должен визуализировать тег `<button>` и показывать значение свойства `item.DisplayName`.

В последнем теге `<div>` используется другое ключевое слово шаблона Angular с именем `*ngIf`. Это ключевое слово шаблона будет оценивать свойство компонента `SelectedItem` и визуализировать HTML-шаблон только в том случае, если свойство SelectedItem является допустимым объектом. Внутри этого тега мы визуализируем свойства `Id` и `DisplayName` свойства `SelectedItem`. Обратите внимание, что без ключевого слова `*ngIf`, если у класса нет свойства с именем `SelectedItem` и мы пытаемся обратиться к подсвойству, как в `SelectedItem.Id`, среда выполнения Angular выдаст ошибки обращения и не сможет визуализировать страницу. Имейте это в виду, когда создаете более комплексные страницы с Angular. Следите за консолью приложения, когда вносите изменения в шаблоны, так как она будет показывать любые ошибки визуализации и, как правило, укажет на используемый шаблон.

После внесения этих изменений наше приложение Angular будет визуализировать массив элементов `ClickableItem` на HTML-странице.

События

Следующий шаг – подключить DOM-события `onclick` и показать, какой элемент выбран в данный момент. Как и в случае с Aurelia, это потребует небольшой модификации нашего шаблона и добавления обработчика событий `click` в нашу модель. Начнем с обновления шаблона в файле `app.component.html`:

```
<ul>
   <div *ngFor="let item of items"
      (click)="onItemClicked(item)">
      <button class="btn btn-primary">{{item.DisplayName}}</button>
      <br />
   </div>
</ul>
```

Здесь мы просто добавили атрибут `(click) = "onSelect(item)"` в тег `<div>`, используемый для визуализации каждого элемента в DOM. Обратите внимание на небольшое различие между синтаксисом Aurelia и Angular, используемым здесь. Там, где Aurelia применяет `click.delegate`, Angular просто использует `(click)`, заключенный в скобки. Теперь мы можем определить функцию `onSelect` внутри класса `AppComponent`:

```
export class AppComponent {
   ...существующий код
   onItemClicked(item: ClickableItem) {
```

```
      this.SelectedItem = item;
    }
  }
```

Мы определили функцию `onSelect`, у которой есть объект `ClickableItem`, в качестве аргумента. В этой функции мы просто устанавливаем значение свойства `SelectedItem` в нашем классе как элемент, который использовался при возникновении события. Как мы видели в Aurelia, этот обработчик также находится на уровне `AppComponent`, а не на `ClickableItem`, как это было в Backbone. Причина этого опять-таки кроется в том, что управляющий класс, то есть класс, который определяет цикл `*ngFor`, – сам `AppComponent`.

После этого наше приложение запущено и работает на Angular.

Формы

На самом деле Angular предоставляет два разных механизма для привязки свойств компонента к значениям формы. Более простой из этих двух методов – формы, основанные на шаблонах, а второй, несколько более сложный способ построения форм, носит название реактивные формы. По этой причине мы создадим две формы в нашем приложении, чтобы иметь возможность изучить оба этих метода связывания.

Формы шаблонов

Формы шаблонов очень похожи на синтаксис привязки форм Aurelia в том смысле, что мы создаем свойство для нашего компонента и автоматически связываем его с элементом управления вводом формы. После привязки свойства Angular обеспечит автоматическое обновление другого изменения, внесенного либо в самом компоненте, либо пользователем, вводящим данные в форму. Чтобы использовать шаблон формы, нам нужно свойство в нашем компоненте, которое будет связано с элементом управления формы. Это простое свойство:

```
export class AppComponent {
  Title = 'Please select: ';
  items = ClickableItemArray;
  Name = 'Your Name';
  ...существующий код
  onSubmit() {
    console.log(`onSubmit: Name: ${this.Name}`);
  }
}
```

Здесь мы обновили наш класс `AppComponent`, добавив в него свойство `Name`, и установили значение по умолчанию `'Your Name'`. Мы также добавили функ-

цию onSubmit, которая будет записывать значение этого свойства в консоль. После этого мы можем обновить наш HTML-шаблон в файле app.component.html:

```
<h2>Template Form :</h2>
<div class="form-group">
   <label for="inputName">Name :</label>
   <input type="text" class="form-control input-name"
      id="inputName" [(ngModel)]="Name" />
</div>
<button id="submit-button-button" class="submit-button"
   (click)="onSubmit()">Submit</button>
```

Мы добавили тег <h2>, чтобы показать начало формы шаблона. Затем мы создаем тег с классом form-group из Bootstrap, напоминающий аналогичный класс в Backbone и Aurelia.

Далее мы создаем тег <label> и элемент <input>. Обратите внимание, что мы добавили атрибут [(ngModel)] со значением "Name". Этот атрибут будет обработан Angular, и он указывает, что значение элемента <input> привязано к свойству модели с именем "Name". Опять же, это двустороннее связывание, означающее, что свойство компонента Name будет синхронизировано с элементом формы <input>.

Мы также добавили элемент <button> и указали, что обработчик (click) будет вызывать функцию onSubmit для компонента при нажатии кнопки. Наряду с изменениями в классе AppComponent это все, что необходимо для привязки свойств нашего класса к элементам формы.

Обратите внимание, что мы должны включить FormsModule в файл app.module.ts, чтобы сделать возможным использование форм в нашем приложении:

```
import { FormsModule } from '@angular/forms';
   ...существующие импорты
@NgModule({
   ...существующий код
   imports: [
      BrowserModule,
      FormsModule
   ]
...существующий код
})

export class AppModule { }
```

Здесь мы импортировали библиотеку FormsModule из '@angular/forms' и добавили ее в массив import в декораторе NgModule.

Ограничения форм шаблонов

Хотя формы шаблонов действительно очень просты и следуют синтаксису, аналогичному формам Aurelia, вскоре становится трудно обеспечить расширенные функциональные возможности формы, использующей синтаксис шаблона. В качестве примера рассмотрим следующий элемент управления формы:

```
<input type="text" [(ngModel)]="Name" required minlength="4">
```

Здесь у нас есть элемент `input`, который связан со свойством компонента `Name`. Обратите внимание, что у нас есть два дополнительных атрибута, которые на самом деле являются валидаторами полей. Это `required`, который указывает, что данный ввод нельзя оставлять пустым, и `minlengt`, который установлен в значение 4. Таким образом, используя синтаксис шаблона, это поле ввода будет проверено до того, как сама форма станет действительной.

К сожалению, при создании больших форм эти валидаторы могут довольно легко меняться в зависимости от контекста формы или другого ввода, выбранного пользователем. Из-за этих ограничений в использовании синтаксиса шаблонов Angular ввел концепцию реактивных форм.

Реактивные формы

Реактивные формы были введены в фреймворк Angular, чтобы предоставить возможность создавать формы, которые могут изменяться динамически. Другими словами, мы можем на лету изменить проверку достоверности определенного элемента управления формы или пометить некоторые свойства как `required` или `disabled` программным способом. В качестве примера некоторые элементы управления формой могут изменять минимальную длину ввода в зависимости от того, что пользователь выбрал в другом элементе управления. Или некоторые входные данные формы могут или не могут быть обязательными, в зависимости от того, что пользователь выбрал в другой части формы. Перемещая объявление этих форм в компонент, мы можем изменить его с помощью стандартного кода и лучше контролировать свои формы.

Чтобы использовать реактивные формы, нам понадобится импортировать модуль `ReactiveFormsModule` в наш файл `app.module.ts`:

```
import { ReactiveFormsModule } from '@angular/forms';
   ...существующие импорты
@NgModule({
   ...существующий код
   imports: [
      BrowserModule,
      FormsModule,
      ReactiveFormsModule
```

```
    ]
    ...существующий код
})

export class AppModule { }
```

Здесь, как и в случае с `FormsModule`, мы импортировали `ReactiveFormsModule` из `@angular/forms` и включили его в массив `import`.

Использование реактивных форм

Чтобы использовать реактивные формы в нашем `AppComponent`, нам нужно будет установить новое свойство `FormGroup` и включить класс `FormBuilder` как часть нашей функции-конструктора:

```
import { FormBuilder, FormGroup, FormControl }from '@angular/
forms';
...существующий код
export class AppComponent {
   ..существующие свойства
   reactiveFormGroup: FormGroup;
   constructor(private formBuilder: FormBuilder) { }
   ...существующий код
}
```

Здесь мы обновили файл `app.component.ts`, импортировав в него классы `FormBuilder`, `FormGroup` и `FormControl` из фреймворка Angular. Затем мы создаем новое свойство `reactiveFormGroup` типа `FormGroup`. Данное свойство будет содержать внутреннюю структуру нашей реактивной формы и даст нам доступ к каждому из его свойств. Мы также определили функцию-конструктор с одним аргументом `formBuilder` типа `FormBuilder`. Мы будем использовать класс `FormBuilder` для создания нашей внутренней формы.

Angular использует технику, известную как внедрение зависимости, для обеспечения компонентов экземплярами классов, которые им нужны для того, чтобы выполнить свою работу. Мы подробно обсудим концепцию внедрения зависимости в следующей главе, но пока помните, что Angular создаст экземпляр класса `FormBuilder` и предоставит его нам через нашу функцию-конструктор.

Теперь мы готовы построить и использовать наш `FormGroup`:

```
export class AppComponent {
   ...существующий код
   ngOnInit() {
      this.reactiveFormGroup = this.formBuilder.group({
         nameInput: new FormControl({})
      });
```

```
        this.reactiveFormGroup.reset({
          nameInput: 'RF Input'
        });
    }
    ...существующий код
    onSubmitRf() {
      console.log(`onSubmitRf: nameInput:
        ${this.reactiveFormGroup.value.nameInput}`);
    }
}
```

Здесь мы создали новую функцию с именем `ngOnInit`. Это специальная функция, которая может быть введена в любой компонент Angular и будет вызываться, когда компонент инициализируется. Обратите внимание, что в Angular есть важное различие между созданием класса и его инициализацией. Создание класса – это когда Angular впервые создает класс. Как часть этого процесса Angular выяснит, какие классы требуются, как определено в конструкторе, и соответственно предоставит экземпляры этих классов, используя свой внутренний фреймворк внедрения зависимости.

Инициализация класса, однако, происходит непосредственно перед тем, как класс визуализируется в DOM. Классы Angular могут иметь свойства, определенные в HTML-коде, и именно при синтаксическом анализе HTML-элементов происходит инициализация и вызывается функция `ngOnInit`.

Функция `ngOnInit` выполняет два отдельных шага при создании нашей реактивной формы. Первым шагом является создание самой формы с помощью функции `group` из экземпляра класса `FormBuilder`. Чтобы определить свойство реактивной формы, у свойства должно быть имя и значение `FormControl`. Итак, другими словами, мы определяем структуру, которая включает в себя имя каждого элемента управления в нашей форме.

Второй шаг – это определение состояния каждого свойства, что делается путем вызова функции `reset` из недавно созданного класса `FormGroup`. Эта функция сопоставляет имя каждого элемента управления формы с предоставленным значением. Итак, глядя на код, мы определили элемент управления формы `nameInput` через экземпляр `formBuilder` и установили значение этого элемента управления как `'RF  Input'` через функцию `reset`. Обратите внимание, что мы должны сопоставить имя элемента управления формы с именем значения по умолчанию.

Мы также определили функцию `onSubmitRf`, которая будет привязана к кнопке **Отправить** в нашей реактивной форме. Эта функция опрашивает значение элемента управления формы с помощью `reactiveFormGroup`. К каждому входному значению можно получить доступ через свойство формы `value` и имя элемента управления формы.

Таким образом, `reactiveFormGroup.value.nameInput` возвращает значение, которое пользователь установил в самой форме.

После внесения изменений в наш компонент мы можем изменить шаблон формы (в `app.component.html`):

```
<h2>Reactive Form :</h2>
<form [formGroup]="reactiveFormGroup" (ngSubmit)="onSubmitRf()">
   <div class="form-group">
      <label for="rfInputName">RF Name :</label>
      <input type="text" class="form-control input-name"
         id="rfInputName" formControlName="nameInput" />
   </div>
   <button type="submit" class="submit-button">Submit Rf</button>
</form>
```

Вначале у нас идет тег `<h2>`, чтобы показать начало реактивной формы. Затем мы создаем тег `<form>` и указываем два атрибута. Первый атрибут – `[formGroup]`, который использует синтаксис шаблона Angular для привязки формы к экземпляру `FormGroup` с именем `reactiveFormGroup`. Это свяжет эту форму с нашим свойством компонента с тем же именем. Второй атрибут – `(ngSubmit)`, который используется для указания функции, вызываемой при отправке формы. В этом случае мы вызываем функцию `onSubmitRf` в `AppComponent`.

Остальная часть шаблона устанавливает элементы `<label>` и `<input>`, как мы видели ранее. Обратите внимание, однако: новый атрибут элемента `<input>` называется `formControlName`.

Этот атрибут используется Angular для привязки значения элемента управления нашей формы к значению элемента `<input>`. Значение этого атрибута должно соответствовать имени элемента управления формы.

Итак, чтобы использовать реактивные формы в Angular, нам нужны три вещи. Во-первых, нам нужно определить структуру и имена свойств формы и сохранить их в экземпляре класса `FormGroup` в нашем компоненте. Затем нам нужно вызвать функцию `reset` этого экземпляра, чтобы установить значения по умолчанию для каждого элемента управления формы. Наконец, нам нужно привязать `FormGroup` к нашему шаблону, используя атрибут `[formGroup]`, и связать каждый HTML-элемент с соответствующим свойством `FormGroup`, используя атрибут `formControlName`.

Резюмируя

Angular быстро становится популярным фреймворком для разработки крупномасштабных приложений. Таким образом, существует множество дополнитель-

ных библиотек, доступных для улучшения разработки на Angular, в том числе элементы управления Plug-and-Play, графики, элементы управления таблицами данных и т. п. Фреймворк Angular предоставляет все инструменты, необходимые для современного приложения, включая ленивую загрузку, обработку маршрутов, защиту при авторизации и проверку элементов контроля формы среди прочего. Будучи сильно связанным с TypeScript, он позволяет использовать шаблоны объектно-ориентированного проектирования, обобщения, методы асинхронного программирования и внедрение зависимости прямо из коробки. Синтаксис Angular довольно прост, хотя и требует некоторых знаний о своей общей экосистеме, чтобы не совершать ошибок.

Использование React

Еще один фреймворк TypeScript, который мы рассмотрим в этой главе, – это React. Фреймворк React имеет открытый исходный код и изначально был разработан Facebook. Он использует специфический встроенный синтаксис для объединения HTML-шаблонов и кода JavaScript в один файл под названием JSX. В ем нет загружаемых шаблонов строк, как в Backbone, или фрагментов HTML-кода, которые находятся в отдельном файле, как в Angular или Aurelia. В React все шаблоны смешаны с обычным кодом JavaScript, используя HTML-подобный синтаксис. В качестве простого примера этого синтаксиса рассмотрим следующий код:

```
render() {
   return <div>Hello <span>React</span></div>;
}
```

Здесь у нас есть стандартная TypeScript-функция `render`. В рамках этой функции мы возвращаем то, что выглядит как нативный HTML-код с тегами `<div>` и дочерними тегами `<span>`. Обратите внимание, что вокруг этих HTML-элементов нет кавычек. Они написаны внутри нашей функции без четкого выделения из обычного кода TypeScript.

TypeScript включил поддержку уникального синтаксиса React/JSX в релизе 1.6. Однако для использования нового синтаксиса JSX нам потребуется создать файлы TypeScript с расширением `.tsx` вместо обычного расширения `.ts`. Когда TypeScript находит файлы с расширением .tsx, он анализирует файл как JSX-файл, что позволяет использовать синтаксис JSX.

Настройка React

Процесс, который React использует для генерации JavaScript из файлов JSX, создает дополнительный шаг в обычном рабочем процессе разработки. Наши файлы TypeScript `.tsx` после компиляции будут генерировать файлы JavaScript, которые преобразуют синтаксис JSX в серию обращений к библиотекам React. Напри-

мер, использование элемента в нашем файле .tsx создаст вызов React.createElement("div", ...) в скомпилированном файле JavaScript. Затем эти скомпилированные файлы необходимо объединить с самими библиотеками React, чтобы создать исполняемый код. По этой причине рекомендуется использовать такой инструмент, как Webpack, чтобы объединить вывод шага компиляции с библиотеками React. Webpack также создаст один выходной файл для загрузки в браузер в процессе, называемом пакетированием.

Чтобы начать новый проект React, мы выполним несколько шагов. Во-первых, создадим каталог для своего проекта и инициализируем npm:

```
mkdir react-sample
cd react-sample
npm init
```

Здесь мы создаем каталог для своего проекта, переходим в него и инициализируем npm в каталоге проекта. После этого будет создан файл package.json, который может использовать npm. После инициализации мы можем установить webpack:

```
npm install -g webpack
npm install -g webpack-cli
```

Будет установлен webpack в качестве глобального модуля Node и интерфейс командной строки webpack, webpack-cli. Обратите внимание, что хотя webpack у нас установлен глобально, инструмент командной строки webpack все равно должен находить модули webpack в каталоге node_modules. Это означает, что нам также необходимо установить webpack в качестве локального модуля:

```
npm install webpack --save-dev
npm install webpack-cli -save-dev
```

Теперь мы можем установить React:

```
npm install react react-dom
```

После этого в каталог node_modules будут установлены библиотеки react и react-dom. Нам понадобится ряд других утилит, а именно:

```
npm install --save-dev ts-loader source-map-loader
```

После этого будут установлены утилиты ts-loader и source-map-loader в качестве зависимостей разработки. Нам также нужно будет установить Boostrap, как мы это делали в наших предыдущих проектах:

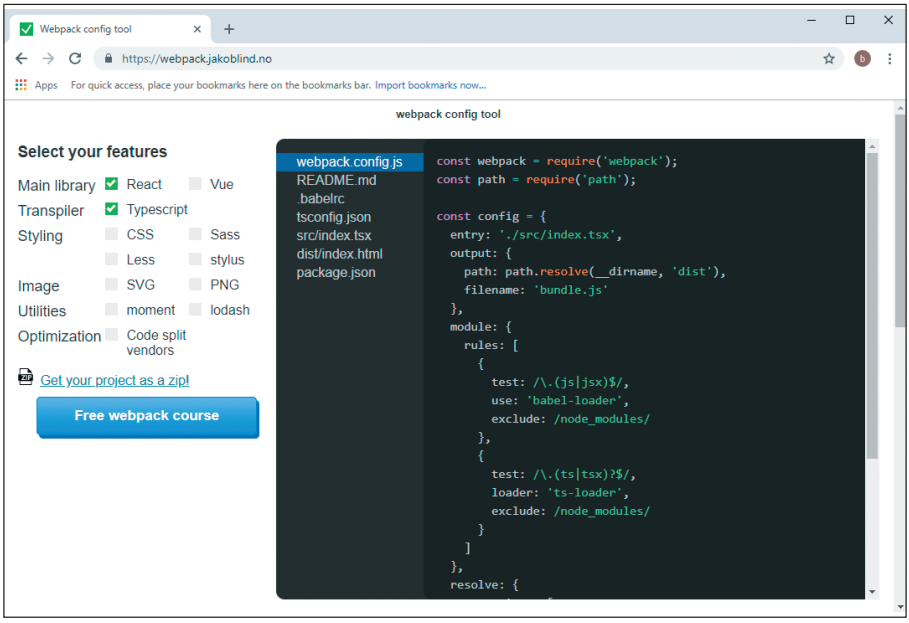
```
npm install bootstrap
```

После установки Bootstrap мы можем установить файлы объявлений react с помощью синтаксиса `@types`:

```
npm install @types/react --save-dev
npm install @types/react-dom --save-dev
```

Настройка webpack

Как было упомянуто ранее, webpack используется для объединения вывода TypeScript с библиотеками React и создания единого объединенного файла JavaScript, который можно использовать в браузере. Для этого, однако, параметры компиляции для TypeScript в файле `tsconfig.json` и конфигурация webpack в файле `webpack.config.js` должны быть согласованы. Самый простой способ сделать это правильно – использовать один из удобных инструментов настройки, доступных в интернете. Один из таких инструментов можно найти на странице `https://webpack.jakoblind.no`. Он предоставляет простую HTML-страницу для настройки ряда параметров, доступных для webpack:



Здесь мы указали, что используем React в качестве главной библиотеки и TypeScript в качестве транспайлера. В правой части страницы видно, что этот инструмент генерирует правильный файл `webpack.config.js`, а также соответствующий файл `tsconfig.json`. Он также создаст образец файла `index.html` в каталоге `dist` и файл `index.tsx` в каталоге `src`. Мы можем либо вручную создать каждый из этих файлов в своем проекте, либо скачать zip-файл, который легко извлечь. Давайте посмотрим на файл `tsconfig.json`:

```
{
   "compilerOptions": {
      "outDir": "./dist/",
      "sourceMap": true,
      "strict": true,
      "noImplicitReturns": true,
      "noImplicitAny": true,
      "module": "es6",
      "moduleResolution": "node",
      "target": "es5",
      "allowJs": true,
      "jsx": "react",
   },
   "include": [
      "./src/**/*"
   ]
}
```

В этом файле установлены три интересных свойства, которые мы не обсуждали ранее. Во-первых, это использование свойства `outDir`, чтобы определить, куда должны быть записаны наши скомпилированные файлы TypeScript. В этой конфигурации свойство outDir установлено в "`./dist./`", чтобы все сгенерированные файлы `.js` были записаны в данный каталог. Второе важное свойство называется "`jsx`" и имеет значение "`react`". Это опция компилятора, которая скажет TypeScript обращаться с файлами `.tsx`, используя синтаксис JSX, и будет генерировать вызовы React для JSX, который включен в наши файлы. Третье важное свойство в этом файле – это свойство `include`, указывающее "`./src/**/*`". Это позволяет webpack знать, что он должен искать в каталоге `src` и всех его подкаталогах все входные файлы `.tsx`.

Последний файл конфигурации, который нам нужен для нашей среды разработки, – это файл `webpack.config.js`:

```
const webpack = require('webpack');
const path = require('path');

const config = {
   entry: './src/index.tsx',
   output: {
      path: path.resolve(__dirname, 'dist'),
      filename: 'bundle.js'
   },
   module: {
      rules: [
         {
            test: /\.(js|jsx)$/,
            exclude: /node_modules/,
            use: 'babel-loader'
         },
```

```
        {
          test: /\.(ts|tsx)?$/,
          loader: 'ts-loader',
          exclude: /node_modules/
        }
      ]
    },
    resolve: {
      extensions: [
      '.js',
      '.jsx',
      '.tsx',
      '.ts'
      ]
    }
}

module.exports = config;
```

Этот файл не требует пояснений, поэтому мы не будем здесь описывать каждую опцию. Однако одно из свойств, которое следует отметить, – это свойство "`entry`" в верхней части файла, указывающее файл "`./src/index.tsx`". Файл `index.tsx` будет использоваться в качестве исходного файла запуска для процесса начальной загрузки React. Другое свойство, которое нужно отметить, – это свойство "`output`", указывающее, что результаты процесса связывания должны быть записаны в файл "`./dist/bundle.js`".

После этого что нам нужно сделать, так это связать нашу глобальную версию TypeScript с локальным проектом:

```
npm link typescript
```

Это позволит webpack использовать нашу глобальную версию компилятора TypeScript в шаге компиляции.

Имея эти файлы конфигурации, мы можем скомпилировать и связать весь код, просто набрав следующее:

```
webpack
```

Запуск `webpack` на этом этапе приведет к ошибке, так как мы еще не создали файл `src/index.tsx`. Чтобы устранить эту ошибку, просто создайте пустой файл `index.tsx` в каталоге `src`.

ItemView

Модели React, как и модели Aurelia и Angular, являются простыми классами TypeScript. Поскольку мы рассматривали эти модели в предыдущих разделах, мы сразу же перейдем к тому, как React создает представления.

Вначале давайте создадим исходный файл `src/ReactApp.tsx`. В этом файле мы создадим два представления. Первое будет представлением для каждого отдельного элемента `ClickableItemArray` и называться `ItemView`. Второе представление будет называться `ItemCollectionView` и визуализировать весь массив. Это похоже на те два представления, которые мы создавали в Backbone. Мы начнем с `ItemView`:

```
import * as React from 'react';
import * as _ from 'underscore';

export class ItemModel {
   DisplayName: string = "";
   Id: number = 0;
   onItemClicked(item: ItemModel) { }
}

export class ItemView
   extends React.Component<ItemModel, {}> {
      constructor(input: ItemModel) {
         super(input);
         this.handleClick = this.handleClick.bind(this);
      }
      render() {
         return (
            <div>
               <button className="btn btn-primary"
                  style={{ "marginBottom": "5px" }}
                  onClick={this.handleClick}>
                  {this.props.DisplayName}
               </button>
            </div>
         );
      }
      handleClick() {
         this.props.onItemClicked(this.props);
      }
   }
```

Мы начинаем с оператора `import`, чтобы импортировать все классы из модуля '`react`', и зададим пространство имен React для этих импортов. Аналогичным образом мы импортируем все классы из модуля '`underscore`' и указываем про-

странство имен _ для этих импортов. Опять же, эти операторы импорта являются частью синтаксиса модуляризации, который мы рассмотрим в `главе 10` *«Модуляризация»*. Затем мы определяем модель `ItemModel`, которая будет нашей моделью данных, как мы делали это ранее при работе с Aurelia и Angular.

Однако обратите внимание, что `ItemModel` определяет свойства `Id` и `DisplayName`, а также функцию `onItemClicked`. Эта функция будет использоваться чуть позже и фактически является функцией обратного вызова, которую родительское представление определит, чтобы мы могли запустить событие в родительском классе при нажатии на один из элементов.

Данный фрагмент кода также определяет класс с именем `ItemView`, который является преставлением React. Оно будет использоваться для визуализации каждого элемента нашего массива. Подобно Backbone, мы определяем представление для каждого элемента массива, вместо того чтобы использовать циклическую конструкцию, такую как `*ngFor`, которую мы видели в Angular.

В React все представления называются модульными, компонуемыми компонентами, и поэтому класс `ItemView` наследует от базового класса `React.Component` или расширяет его. Этот базовый класс использует синтаксис обобщений для определения двух обязательных параметров обобщенного типа. Первый параметр – это модель, к которой относится представление, в данном случае это модель `ItemModel`. Второй объект – это состояние по умолчанию, в котором будет работать эта модель. Поскольку на данном этапе нам не требуется состояние по умолчанию, мы оставим это как пустой объект.

У `ItemView` есть функция-конструктор, функция `render` и функция `handleClick`. Функция-конструктор начинается с вызова `super`, чтобы инициировать базовый класс `React.Component`, а затем устанавливает функцию `handleClick` в результат вызова функции `bind` для функции `handleClick` с `this` в качестве параметра. Это довольно странное выражение на самом деле функционально эквивалентно вызову `_.bindAll(this, 'handleClick')`. Это подчеркнутый способ гарантировать, что экземпляр класса используется всякий раз, когда вызывается функция `handleClick`.

Функция `render` несколько интереснее.

Все функции render в React должны возвращать фрагмент HTML-кода. Однако при более внимательном рассмотрении этого фрагмента мы заметим, что он не обрабатывается как строка. Другими словами, компоненты React могут включать HTML-элементы в свои функции render, как если бы они были частью стандартного языка. Эта особенность является причиной, по которой файлы React должны определяться с использованием специального расширения `.tsx`. Используя расширение `.tsx`, мы уведомляем компилятор TypeScript о том, что смешиваем собственный HTML-код и код TypeScript в одном и том же файле.

Функция render возвращает тег , а внутри него тег <button>. В теге <button> виден синтаксис шаблонов React, который используется для внедрения свойства модели DisplayName через синтаксис {this.props.DisplayName}. React позволяет получить доступ к свойствам базовой модели через свойство класса props.

Обратите внимание, что мы также определили имя класса CSS для этого компонента, "btn btnprimary", но имя этого атрибута – className, а не просто class, потому что React использует HTML-атрибуты в качестве хуков в свойствах стандартного класса. Другими словами, чтобы использовать атрибут myAttribute в рамках фрагмента React JSX, у класса, к которому мы обращаемся, должно быть определенное свойство myAttribute. По этой причине React переименовал общие HTML-атрибуты, такие как class, в className.

Обратите внимание, что мы определили атрибут стиля, применяя двойные круглые скобки, {{ и }}. Этот стиль не использует стандартное свойство CSS margin-bottom, но использует переименованное свойство marginBottom. Это еще один пример того, что React нужно слегка переименовать некоторые из наиболее распространенных свойств CSS во избежание коллизий. Полный список свойств CSS, которые были переименованы, см. в определении StandardLonghandProperties в файле объявлений React.

Последняя функция в этом представлении – это функция handleClick. Она вызывает функцию onItemClicked, которая является частью исходной модели, использовавшейся при создании этого представления. Она вызывает эту функцию с внутренней моделью, к которой обращаются через this.props. Помните, что React хранит значения свойств модели внутри себя, и к этим значениям можно обратиться в коде, используя внутреннее свойство props.

CollectionView

Второе представление, которое мы определим, – это представление, которое будет использоваться для визуализации всего ClickableItemArray. Как упоминалось ранее, React похож на Backbone в том смысле, что мы определим представление для каждого отдельного элемента массива (ItemView), а затем еще одно представление для всей коллекции. Это представление будет называться ItemCollectionView и использовать два интерфейса:

```
export interface IClickableItem {
   Id: number;
   DisplayName: string;
}

export interface IItemCollectionViewProps {
   title: string,
```

```
    items: IClickableItem[],
    SelectedItem: IClickableItem;
};
```

Здесь мы определили интерфейс для каждого из наших элементов массива с именем `IClickableItem`, у которого есть свойства `Id` и `DisplayName`. Наш второй интерфейс называется `IItemCollectionViewProps` и определяет свойства, которые потребуются нашему представлению коллекции. Свойство `Title` будет содержать фразу `"Please select"`, свойство `items` будет содержать наш массив, а свойство `SelectedItem` – текущий выбранный элемент.

Обратите внимание, что компоненты React всегда создаются путем расширения `React.Component` с использованием синтаксиса обобщений и передачи типа, который определяет свойства компонента. Это означает, что мы не можем просто создать свойство для компонента React и ожидать, что оно будет частью собственности `props`. Давайте посмотрим на определение этого представления:

```
export class ItemCollectionView extends
    React.Component<IItemCollectionViewProps, {}> {
    // SelectedItem: IClickableItem;
    // ^^ Свойства нельзя подключать подобным образом;
    constructor(input: IItemCollectionViewProps) {
        super(input);
        this.itemSelected = this.itemSelected.bind(this);
        // ^^ Это функционально эквивалентно приведенной ниже строке:
        // _.bindAll(this, 'itemSelected');
    }
    ... другой код
}
```

Здесь мы определили класс `ItemCollectionView`, который расширяет `React.Component`. Опять же, синтаксис обобщений для компонентов React требует двух аргументов: объекта, описывающего свойства представления, и объекта, описывающего его начальное состояние. Когда мы передаем интерфейс `IItemCollecitonViewProps` в синтаксисе обобщений, мы знаем, что этот компонент React будет иметь свойства `Title`, `items` и `SelectedItem`. Конструктор вызывает super, как было показано ранее, а затем связывает контекст выполнения функции `itemSelected` с экземпляром класса. Как уже было отмечено, функция с подчеркиванием `bindAll` является функциональным эквивалентом.

Давайте теперь посмотрим на функцию `render`:

```
render() {
    let _this = this;
    let buttonNodes = this.props.items.map(function (item) {
        return (
            <ItemView onItemClicked={_this.itemSelected}
```

```
            DisplayName={item.DisplayName}
            Id={item.Id}
          />
        );
      });
      return <div>
        <h1>{this.props.title}</h1>
        <ul>
          {buttonNodes}
        </ul>
        <div>Selected Item :
          {this.props.SelectedItem.Id} -
          {this.props.SelectedItem.DisplayName}</div>
      </div>;
  }
```

Вначале мы видим старую уловку JavaScript, использованную для определения локальной переменной `_this`, которая установлена в `this`. Причина, по которой мы это делаем, заключается в том, что мы можем сослаться на экземпляр класса `ItemCollectionView` чуть позже. Функция `render` разделена на две части.

Первая часть – это определение имени переменной, `buttonNodes`, которая вызывает функцию `map` для свойства `items`, как видно из вызова `this.props.items.map`. Эта функция будет перебирать каждый элемент массива `items` и возвращать фрагмент HTML-кода. Возвращенный фрагмент – это HTML-элемент с именем `ItemView`. Это имя совпадает с именем класса `ItemView`, который мы определили ранее. У этого элемента есть HTM-атрибут `onItemClicked`, а затем атрибуты `DisplayName` и `Id`, которые необходимы `ItemView`. Свойство `onItemClicked` использует локальную переменную `_this` для передачи функции `itemSelected` в качестве функции обратного вызова в `ItemView`. Следовательно, функция `itemSelected` класса `ItemCollectionView` будет вызываться (посредством функции обратного вызова) при щелчке на `ItemView`.

Вторая часть функции `render` возвращает фрагмент HTML-кода для всего `ItemCollectionView`. Она состоит из тега `<h1>`, который визуализирует свойство `title` с помощью синтаксиса `{this.props.title}`. Следующий тег – это `<ul>`, а внутри него – переменная `{buttonNodes}`. Таким образом, этот фрагмент будет включать в себя результаты нашей функции `map`, как это определено переменной `buttonNodes`. Итак, React позволяет нам использовать любую переменную или функцию TypesScript в наших HTML-шаблонах простым и интуитивно понятным способом. Последняя часть этого фрагмента кода будет визуализировать выбранный в данный момент элемент внутри тега `<div>`.

Есть еще одна функция, которую мы должны определить в классе `ItemCollectionView`:

```
itemSelected(item: IClickableItem) {
   this.props.SelectedItem.Id = item.Id;
   this.props.SelectedItem.DisplayName = item.DisplayName;
   this.setState({});
}
```

Функция `itemSelected` будет использоваться в качестве функции обратного вызова при нажатии `ItemView`. У этой функции есть единственный параметр `item` типа `IClickableItem`. Таким образом, сигнатура функции соответствует определению функции `onItemClicked` в `ItemModel`, которую мы использовали для визуализации каждого элемента коллекции. Эта функция устанавливает значение свойств `Id` и `DisplayName` внутреннего свойства `SelectedItem`, чтобы соответствовать входящим значениям. Опять же, чтобы обратиться к внутреннему свойству определенного представления, мы должны использовать свойство `props`. В конце эта функция вызывает функцию `setState` с пустым объектом. Вызов `setState` повторно визуализирует всё представление в DOM.

Начальная загрузка

Чтобы увидеть результаты определений представлений, визуализированных в нашем приложении, нам нужно будет выполнить начальную загрузку своего кода, аналогично тому, как это мы делали, работая с Aurelia и Angular. Для этого мы изменим файл `app/index.tsx`:

```
import * as React from "react";
import * as ReactDOM from "react-dom";

import {ItemCollectionView, IClickableItem}
   from "./ReactApp";

let ClickableItemArray: IClickableItem[] = [
   {Id: 1, DisplayName: "firstItem"},
   {Id: 2, DisplayName: "secondItem"},
   {Id: 3, DisplayName: "thirdItem"},
];

ReactDOM.render(
   <ItemCollectionView items={ClickableItemArray}
      title="Please select:"
      SelectedItem={
         {Id: 0, DisplayName: "None Selected "}
      } />,
   document.getElementById("app")
);
```

В начале файла идут стандартные операторы `import`. Первые два оператора делают все классы из библиотеки `"react"` и `"react-dom"` доступными в пространст-

вах имен `React` и `ReactDOM` соответственно. Третий оператор делает класс `ItemCollectionView` и интерфейс `IClickableItem` доступными из файла `ReactApp.tsx`, который мы создали ранее.

Затем мы создаем переменную `ClickableItemArray`, которая является массивом объектов `ClickableItem`. Наконец, мы вызываем функцию `ReactDOM.render`, чтобы визуализировать элемент в DOM типа `ItemCollectionView`. Обратите внимание, что мы указали три атрибута для элемента `ItemCollectionView`.

Первый атрибут – `items`, а значением этого элемента является экземпляр созданного нами массива с именем `ClickableItemArray`. Опять же, React позволяет нам использовать синтаксис `{variableName}` для вставки значения переменной `ClickableItemArray` в DOM. Второй атрибут в этом фрагменте называется `title` и имеет строковое значение `"Please select:"`. Третий атрибут – `SelectedItem`, который представляет начальное состояние свойства `SelectedItem` при первоначальном запуске, а следовательно, `"None Selected"`.

После того как мы определили эти элементы DOM, обратите внимание на то, что мы включаем запятую, а затем вызов в `document.getElementById`.

Этот синтаксис является способом, который использует React для выбора элемента DOM на HTML-странице и внедрения сгенерированного HTML-кода. Идентификатор элемента – `"app"`, и как таковой он будет соответствовать элементу `<div id = "app">` в нашем HTML-коде.

Теперь нам нужно создать файл `index.html`, чтобы выполнить начальный запуск и визуализацию нашего приложения. Этот файл (в корневом каталоге нашего проекта) будет выглядеть так:

```
<!DOCTYPE html>
<html>
<head>
   <title>React starter app</title>
   <link rel="stylesheet" type="text/css"
      href="./node_modules/bootstrap/dist/css/bootstrap.css" />
   <script src="./node_modules/underscore/underscore.js"></script>
</head>
<body>
   <div id="app"></div>
   <script src="./dist/bundle.js"></script>
</body>
</html>
```

Этот HTML-файл содержит тег `<link>` для загрузки файла `bootstrap.css`, а также тег `<script>` для загрузки библиотеки Underscore. Элемент `<body>` определяет тег `<div>` с именем `"app"`, где React будет визуализировать `ItemCollectionView`. В конце мы включаем тег `<script>` для загрузки файла `bundle.js` в каталог `dist`.

Обратите внимание, что поскольку webpack объединяет библиотеки React с нашими скомпилированными классами TypeScript в файл `bundle.js`, этот HTML-файл становится очень простым. Нам не нужно загружать какие-либо специальные библиотеки, так как все уже объединено в один файл `bundle.js` за нас.

Теперь наше приложение React готово к запуску. Просто запустите браузер и откройте файл `index.html` в своем исходном каталоге. Обратите внимание, что если Aurelia и Angular требуют, чтобы веб-сервер работал в среде разработки, React и Backbone таких требований не выдвигают.

Формы

Формы в React строятся вокруг концепции состояния формы. Там, где мы можем связать свойства компонента React, используя свойство `props`, когда мы имеем дело с формами, нам нужно работать со свойством `state`. Это свойство будет содержать набор подсвойств, которые будут связаны с нашими элементами управления формой. В качестве примера рассмотрим следующий код:

```
export class ItemCollectionView
   ...код существующего класса
{
   constructor(input: IItemCollectionViewProps) {
      super(input);
      this.state = {inputName: 'Your Name'};
      ...код существующего конструктора
   }
```

Здесь мы добавили одну строку в функцию-конструктор нашего компонента `ItemCollectionView`, которая присваивает объект свойству `state` компонента `React`. Содержимое этого объекта – просто свойство `inputName`, которое имеет значение 'Your Name'. Этот метод заключается в том, что мы создаем элементы управления форм в React и устанавливаем их значения по умолчанию. Мы можем привязать это значение к нашему шаблону React следующим образом:

```
render() {

...существующий код
return <div>
   <h1>{this.props.title}</h1>
   ...существующий код
   <form>
      <div className="form-group">
         <label>Name :</label>
         <input type="text" className="form-control"
            value={this.state.inputName} />
      </div>
```

```
        <button className="submit-button"
          type="submit" value="Submit">Submit</button>
      </form>
    </div>;
    }
```

Здесь мы изменили функцию `render` для компонента `ItemCollectionView`.

Мы добавили тег `<form>` с тегами `<label>` и `<input>`, как мы это делали, работая с другими фреймворками. Обратите внимание, что у тега `<input>` есть атрибут `value`. Этот атрибут напрямую обращается к свойству `inputName` переменной `state` в нашем компоненте. Помните, что мы устанавливаем значение этого свойства в нашем конструкторе, поэтому если мы сейчас запустим нашу страницу, то увидим значение элемента управления вводом, установленное как `'Your Name'`. Пока все идет нормально.

К сожалению, если мы попытаемся изменить значение элемента управления вводом на странице, то увидим, что он доступен только для чтения. Хотя значение по умолчанию установлено правильно, наш пользователь не может его изменить. Теперь нам нужно присоединиться к жизненному циклу изменения состояния React.

Изменение состояния

Чтобы обнаружить изменение в состоянии элемента управления в React, нам нужно будет создать функцию, которую можно вызывать при каждом изменении:

```
onChange(event: React.ChangeEvent<HTMLInputElement>) {
    let valueName = event.target.name;
    this.setState({ [valueName]: event.target.value });
    console.log(`onChange : ${event.target.name} :
      ${event.target.value}`);
}
```

Здесь мы создали функцию `onChange` с единственным аргументом `event` типа `React.ChangeEvent`. Внутри этой функции мы находим имя HTML-элемента через `event.target.name`, а затем вызываем функцию `setState` с обновленным значением (найденным через `event.target.value`). Поэтому эта функция обновит свойство `state` для нашего компонента. Обратите внимание, что мы также записываем сообщение в консоль, чтобы показать, какими были значения `event.target.name` и `event.target.value`.

Есть две проблемы с этим кодом. Во-первых, как мы узнаем, что свойство `name` в `event.target.name` будет соответствовать внутреннему свойству нашего объекта `state`? Ответ состоит в том, чтобы обновить шаблон и указать HTML-атрибут `name` для нашего элемента управления:

```
<div className="form-group">
   <label>Name :</label>
   <input type="text"
      className="form-control"
      name="inputName"
      value={this.state.inputName}
      onChange={this.onChange}
   />
</div>
```

Здесь мы обновили элемент управления <input> двумя дополнительными атрибутами. Первый – атрибут name, для которого установлено значение inputName. Это гарантирует, что функция onChange получит строку "inputName" в свойстве event.target.name. Второй атрибут, который мы включили, – это обработчик onChange, который будет вызывать функцию onChange в нашем компоненте при изменении значения. Элемент управления вводом теперь подключен для отправки правильной информации в обработчик onChange.

Свойства состояния

Однако если мы попытаемся скомпилировать наш код на этом этапе, то получим ошибку компиляции:

```
TS2345: Argument of type '{ [x: string]: string; }' is not
assignable to parameter of type '{ inputName: string; }'
```

Эта ошибка высвечивает вторую проблему в функции onChange, которая вызвана следующей строкой кода:

```
this.setState({ [valueName]: event.target.value });
```

Здесь мы вызываем функцию setState с объектом, который использует свойство индекса: [valueName], что во время выполнения будет эквивалентно [inputName]: event.target.value.

Хотя это допустимый код TypeScript и мы можем использовать этот метод для присваивания значения свойства, помните, что компонент React использует синтаксис обобщений для определения как внутренних свойств компонента, так и свойств state. На самом деле причина этой ошибки – определение компонента React:

```
export class ItemCollectionView extends
   React.Component<
      IItemCollectionViewProps, // props properties
      {inputName: string} // state properties
   > {
   ...существующий код
}
```

Здесь видно, что мы создаем компонент `ItemCollectionView` и наследуем (или расширяем) его из `React.Component`. `React.Component` использует два обобщенных свойства в своем определении. Первое свойство определяет внутреннюю структуру props и имеет значение `IItemCollectionViewProps`. Второе свойство определяет структуру свойства состояния и имеет значение `{inputName: string}`. Это определение и вызывает ошибку компиляции.

Чтобы исправить ее, мы можем обновить определение:

```
{inputName: string, [key: string]: any}
```

Определение теперь включает в себя свойство индекса типа string: `[key: string]`. Поскольку мы не знаем, какой тип будет возвращать это свойство, мы можем оставить тип `any`. Хотя это одно из решений нашей проблемы, мы также можем использовать отображаемый тип, чтобы добиться того же:

```
type StringProps<T> = {
   [key: string]: any;
}

export class ItemCollectionView extends
   React.Component<
      IItemCollectionViewProps, // internal properties
      StringProps<{ inputName: string }> // state properties
   > {
   ...существующий код
}
```

Здесь мы определили отображаемый тип `StringProps`, который использует синтаксис обобщений для типа `T`. Этот тип просто определяет свойство индекса типа `string` для нашего типа. Обновленная версия определения для класса `ItemCollectionView` теперь использует `StringProps<{inputName: string}>`. Эффект тот же, что и в нашем предыдущем решении. Однако преимущество этого отображаемого типа состоит в том, что мы можем повторно использовать его для любого компонента React, и это делает наш код несколько чище.

Теперь мы можем прикрепить обработчик к самой форме:

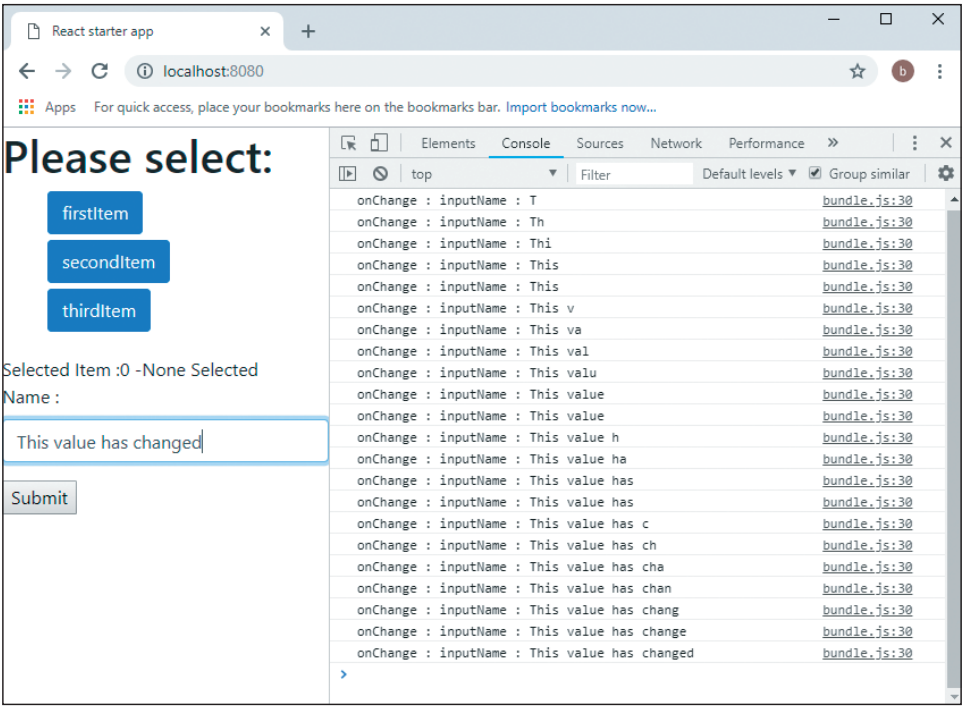
```
render() {
...существующий код
   return <div>
   ...существующий шаблон
      <form onSubmit={this.onSubmit} >
      ... существующий шаблон
      </form>
   </div>;
}
```

```
onSubmit(e: React.FormEvent) {
   console.log(`onSubmit : state : ${this.state.inputName}`);
   e.preventDefault();
}
```

Мы обновили функцию `render` и добавили обработчик `onSubmit` к самой форме. Обработчик вызовет функцию `onSubmit`, которая будет просто записывать значение свойства `inputName` объекта `state` в консоль. Обратите внимание, что мы вызываем `e.preventDefault()`, чтобы убедиться, что не отправлена сама HTM-страница, что вызвало бы полное обновление страницы.

После этих изменений наша форма React готова. Мы видели, как использовать объект состояния для хранения переменных формы и как прикрепить эти переменные к нашим элементам управления вводом.

Мы также обсудили жизненный цикл обнаружения изменений в React и внесли некоторые изменения в наше определение компонента React, чтобы правильно использовать эти события. Если мы запустим наше приложение сейчас и начнем вводить текст в элемент управления формы, обратите внимание, что происходит в консоли:



Здесь видно, что событие `onChange` запускается для каждого нажатия клавиши, которое вводит пользователь. Это немного отличается от других фреймворков,

с которыми мы работали, где эти изменения, по существу, скрыты, пока не потребуется значение элемента управления вводом.

Возможности TypeScript в React

TypeScript поддерживает синтаксис JSX, который React использует в течение некоторого времени. За последние несколько выпусков команда также включила ряд дополнительных функций JSX, чтобы несколько упростить программирование с использованием React. Обновления языка TypeScript для React включают в себя синтаксис оставшихся параметров для свойств шаблона и поддержку свойств по умолчанию.

Синтаксис оставшихся параметров

Когда у нас есть несколько свойств, которые включены в компонент, нам нужно назвать каждое свойство как часть нашего шаблона:

```
return (
   <ItemView
      onItemClicked={_this.itemSelected}
      DisplayName={item.DisplayName}
      Id={item.Id}
   />
);
```

Здесь мы создаем компонент `ItemView` в функции `render`. Компоненту нужны три свойства: `onItemClicked`, `DisplayName` и `Id`. Когда список свойств компонента становится большим, он получается очень утомительным и подвержен ошибкам, если приходится называть каждое свойство в нашем шаблоне.

TypeScript теперь поддерживает использование синтаксиса оставшихся параметров в этих случаях:

```
return (
   <ItemView
      onItemClicked={_this.itemSelected}
      {...item}
   />
);
```

Мы создаем компонент `ItemView` в нашем шаблоне как обычно и указываем свойство `onItemClicked`. Затем мы используем синтаксис оставшихся параметров, `{... item}`, чтобы автоматически назначать каждое свойство.

Свойства по умолчанию

Еще одна функция, включенная командой TypeScript для React и JSX, – это поддержка свойств по умолчанию. Если мы посмотрим, как создаем `ItemCollectionView`, то увидим, как это может быть полезно:

```
ReactDOM.render(
   <ItemCollectionView items={ClickableItemArray}
      title="Please Select:"
      SelectedItem={
         {Id: 0, DisplayName: "None Selected"}
      }
   />,
   document.getElementById("app")
);
```

Здесь мы создаем экземпляр нашего компонента `ItemCollectionView`. Поскольку у компонента есть три свойства, а именно `items`, `title` и `SelectedItem`, нам необходимо определить эти значения при создании компонента. Хотя это и работает, было бы удобно иметь возможность устанавливать значения этих свойств по умолчанию в самом компоненте. Другими словами, зачем пользователю компонента устанавливать значения компонента по умолчанию? Конечно, сам компонент должен принимать эти решения.

Чтобы удовлетворить этот сценарий, TypeScript внедрил поддержку свойства `defaultProps`:

```
export class ItemCollectionView extends
   React.Component {
   ...существующий код
   static defaultProps = {
      title: "Please select:",
      SelectedItem: {Id: 0, DisplayName: "None Selected"}
   };
   ...существующий код
}
```

Здесь видно, что у класса `ItemCollectionView` теперь есть статическое свойство `defaultProps`. Это свойство содержит свойства `title` и `SelectedItem` и устанавливает для них значения по умолчанию. Следовательно, сам компонент теперь определяет эти свойства по умолчанию. Теперь мы можем удалить их из создания компонента:

```
ReactDOM.render(
   <ItemCollectionView items={ClickableItemArray} />,
   document.getElementById("app")
);
```

Здесь мы не упоминаем свойства `title` или `SelectedItem` при создании компонента и оставляем их значения по умолчанию самому компоненту.

Резюмируя

В этом разделе мы рассмотрели, как создать наше приложение с использованием React. Как мы уже видели, React использует синтаксис JSX, который объединяет объявления в стиле HTML прямо внутри наших классов TypeScript. По этой причине нам нужно использовать расширение `.tsx` для файлов TypeScript вместо стандартного расширения `.ts`. Мы видели, как создавать дочерние компоненты и как компонент React может использовать их в собственной функции визуализации. Мы создали наше приложение и увидели, как использовать массивы компонентов, как обрабатывать события DOM, такие как `onClick`, и как React использует функции обратного вызова для передачи сообщений от дочернего компонента к родительскому.

Мы также изучили обработку форм и увидели, как установить значение по умолчанию для компонента формы в нашем коде и как обрабатывать переменную `state`, чтобы перехватывать изменения в значениях нашей формы. Наконец, мы увидели некоторые улучшения относительно поддержки JSX в рамках компилятора TypeScript, используя синтаксис оставшихся параметров для свойств компонента, и как использовать свойства по умолчанию.

React – очень популярный фреймворк, отличающийся от других фреймворков использованием синтаксиса JSX. Компоненты React должны быть рассчитаны на обработку одного небольшого элемента всего экрана, чтобы эти компоненты можно было повторно использовать, где угодно. Поэтому разработка на React заключается в определении этих компонентов и создании приложения из имеющихся у вас под рукой компонентов.

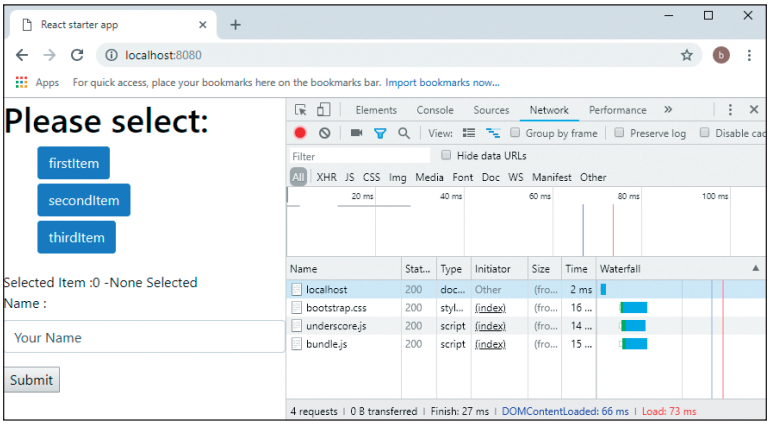
Сравнение производительности

Теперь, когда мы создали одно и то же приложение, используя несколько разных библиотек, совместимых с TypeScript, мы можем сравнить их с точки зрения производительности. Каждый фреймворк загружает HTML-страницу, которая будет загружать необходимые библиотеки вместе с файлами `bootstrap.css` и любыми зависимыми пакетами. Каждое создаваемое нами приложение выполняет одно и то же:

- загружает массив объектов для использования в качестве коллекции по умолчанию;
- визуализирует одно представление для каждого из объектов в коллекции;
- объединяет каждое отдельное представление в представление для всей коллекции;
- визуализирует элементы title и selected item, чтобы показать, какой эле-

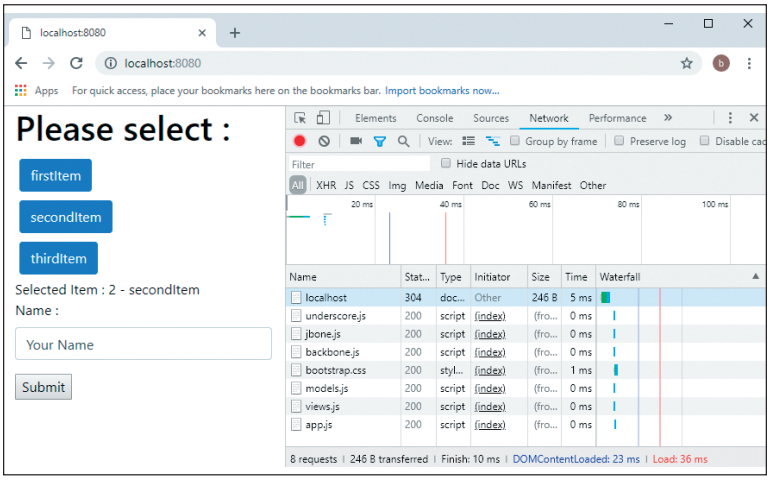
мент выбран в данный момент;
- присоединяет сгенерированный HTML-код к DOM.

Если мы запустим каждую версию этого приложения в одном и том же браузере и откроем наши удобные инструменты для разработчиков, то сможем начать сравнивать, сколько времени уходит у каждого из этих фреймворков, чтобы визуализировать наше приложение в браузере, как показано на приведенном ниже скриншоте:

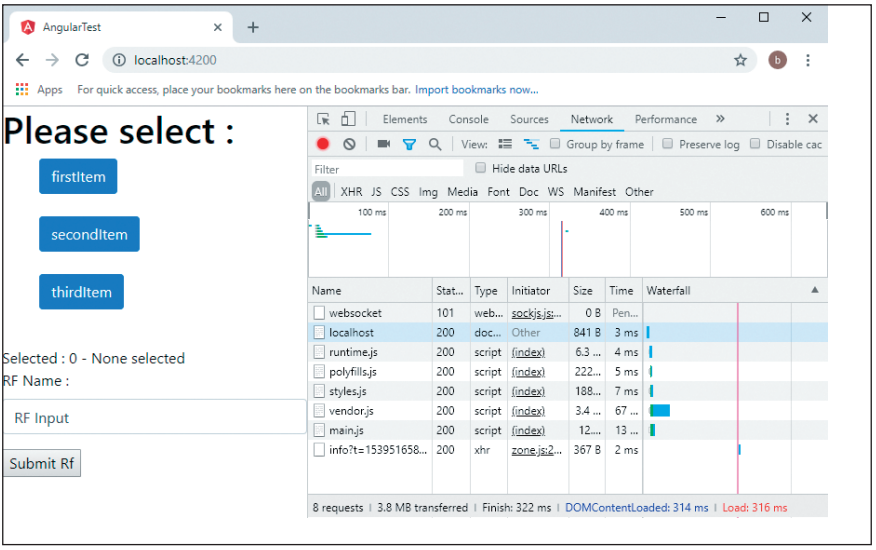


В правом нижнем углу мы видим три числа, представленных в миллисекундах. Важным является номер загрузки, который представляет собой количество времени, которое потребовалось для загрузки всей страницы и ее визуализации в DOM. Как видно из этого скриншота, странице потребовалось 73 миллисекунды. Это версия нашего приложения для React.

Далее идет Backbone, который загружается примерно за 36 миллисекунд:

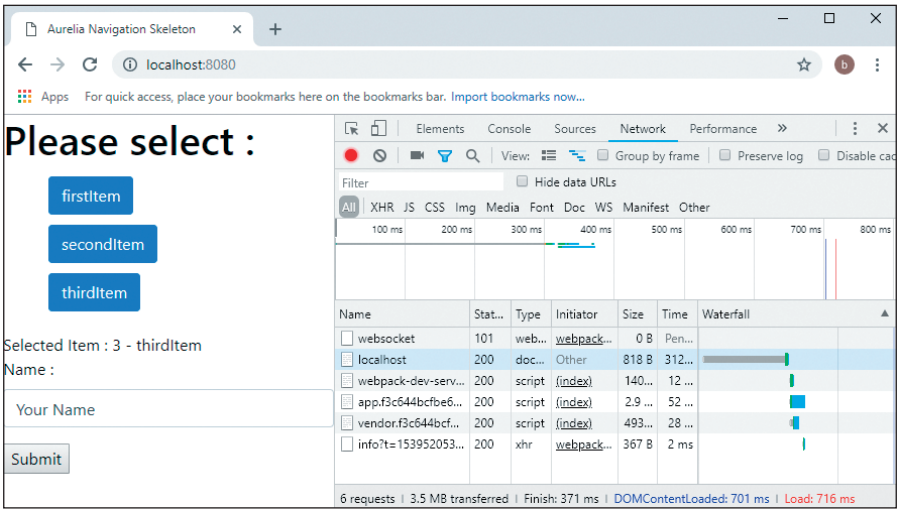


Backbone визуализирует страницу менее чем за половину времени, затрачиваемого для React. Так как же обстоят дела с Angular?



Здесь видно, что Angular визуализирует страницу примерно за 316 миллисекунд, что в 10 раз медленнее, чем у Backbone, и в пять раз медленнее, чем у React.

И наконец, Aurelia:



Aurelia требуется 716 миллисекунд для загрузки и визуализации страницы. Что касается производительности, то на первом месте стоит Backbone, который загружает и визуализирует страницу за 36 миллисекунд. На втором месте – React,

у которого уходит на это 73 миллисекунды, что вдвое больше, чем у Backbone. На третьем месте находится Angular, у которого уходит 316 миллисекунд, и на последнем месте – Aurelia, с 716 миллисекундами.

Таким образом, из этих результатов видно, что чем больше тяжелой работы делает за вас фреймворк, тем больше обработки он будет выполнять в фоновом режиме и тем медленнее будет время загрузки страницы. Сравнивая количество кода, которое нам нужно было написать для Backbone и Aurelia, очевидно, что Aurelia проделывает много работы в фоновом режиме, выявляя, когда нам нужно повторно визуализировать элемент, и автоматически обновляя DOM. Хотя такого рода функциональность прекрасна с точки зрения разработки, это может повлиять на производительность визуализации.

Как упоминалось ранее в этой главе, при нацеливании на старые процессоры и более медленных сетевых подключениях может потребоваться обеспечить максимально быструю визуализацию приложения. Это особенно актуально, если вы ориентируетесь на старые браузеры в старых телефонах в регионах, где покрытие сети медленное или периодически прерывается. Но в конце концов, это вопрос личного выбора. Если Aurelia помогает вашей команде создавать и доставлять приложения с очень высокой скоростью из-за небольшого объема кода, то скорость визуализации приложения может быть не так важна.

Резюме

В этой главе мы подробно рассмотрели, что такое MVC-фреймворк, и обсудили каждый из его элементов. Мы разобрали роль и обязанности модели, представления и контроллера в MVC и то, как они взаимодействуют друг с другом при создании пользовательских интерфейсов. Мы также провели краткое обсуждение преимуществ использования MVC-фреймворков. Затем мы изучили четыре MVC-фреймворка, которые либо очень тесно интегрированы с TypeScript, либо написаны с учетом TypeScript. Мы реализовали одно и то же базовое приложение в каждом из них и сравнили различия в концепциях и синтаксисе между Backbone, Aurelia, Angular и React. Мы также обсудили факторы, влияющие на производительность при работе с каждым из этих фреймворков.

В нашей следующей главе мы рассмотрим автоматизированное тестирование – модульное тестирование, интеграционное тестирование и приемочное тестирование для приложений TypeScript.

Глава 8

Разработка через тестирование

В предыдущей главе мы подробно рассмотрели шаблон проектирования MVC и создали приложения с использованием четырех различных фреймворков, которые используют этот шаблон. Мы увидели, что каждый фреймворк работает немного по-разному с точки зрения моделей и представлений и что в каждом фреймворке есть понятие компонента контроллера или прикладного компонента. Основные принципы шаблона проектирования MVC породили другие аналогичные шаблоны, например **Model View Presenter** (**MVP**) и **Model View View Model** (**MVVM**). Обсуждая эту группу шаблонов вместе, некоторые описывают их как **Model View Whatever** (**MVW**), или **MV***.

Некоторые из преимуществ стиля написания приложений MV* включают в себя модульность и разделение задач, о которых мы расскажем в следующих главах. Но этот стиль также имеет огромное преимущество – возможность писать тестируемый JavaScript-код.

Использование MV* позволяет проводить модульное, интеграционное и приемочное тестирование практически всего нашего прекрасно написанного вручную JavaScript-кода. Мы можем написать тесты для отдельных классов, а затем расширить эти тесты, чтобы охватить группы классов. Мы можем протестировать свои модели, но также можем проверить представления наряду с нашими функциями визуализации – для обеспечения правильного отображения элементов DOM на странице. Мы также можем имитировать нажатия кнопок, выпадающие списки и анимацию. Затем эти тесты можно распространить на переходы страниц, включая страницы входа и домашние страницы. Создав большой набор тестов для своего приложения, мы обретем уверенность в том, что наш код работает должным образом, и позволим себе в любое время перепроектировать его.

Рефакторинг – это возможность изменять код, не опасаясь, что изменится общая функциональность. Это означает, что если у нас есть набор тестов, мы можем переписать любую часть базового кода, пока идет тестирование. Существует старая поговорка, что без тестов вы не перепроектируете, а просто случайным обра-

зом меняете вещи. В большом теле кода даже изменения в одной строке могут иметь нежелательные побочные эффекты, которые нелегко найти, если у вас нет тестов.

В этой главе мы рассмотрим разработку через тестирование применительно к TypeScript. Мы обсудим некоторые из наиболее популярных фреймворков тестирования, напишем несколько модульных тестов с использованием этих фреймворков, а затем обсудим библиотеки для модульного тестирования и методы непрерывной интеграции.

Темы, которые мы рассмотрим в этой главе:

- разработка через тестирование;
- модульные, интеграционные и приемочные тесты;
- Jasmine;
- библиотеки Jasmine для модульного тестирования;
- автоматизация браузера;
- непрерывная интеграция.

Разработка через тестирование

Разработка через тестирование – это способ размышлять над нашим кодом, который должен быть частью стандартного процесса разработки. Это парадигма разработки, которая начинается с тестов и управляет движущей силой производственного кода посредством этих тестов. Разработка через тестирование – это как задать вопрос: *откуда мне знать, что я решил проблему?* а не просто: *как мне решить проблему?* Это важная идея, чтобы понять. Мы пишем код для решения проблемы, но мы должны быть в состоянии доказать, что решили проблему с помощью автоматизированных тестов.

Основными этапами этого подхода являются:

- написание теста, который не проходит;
- запуск теста, чтобы убедиться, что он не проходит;
- написание кода для прохождения теста;
- запуск теста, чтобы убедиться, что он проходит;
- запуск всех тестов, чтобы увидеть, что новый код не нарушает другой;
- повторение.

Использование разработки через тестирование – это действительно образ мышления. Некоторые разработчики следуют этому подходу и сначала пишут тесты, в то время как другие пишут сначала свой код, а потом тесты. Тогда есть те, которые вообще не пишут тесты. Если вы попадаете в последнюю категорию, то надеюсь, что методы, о которых вы узнаете в этой главе, помогут вам начать действовать в правильном направлении.

Есть так много оправданий, чтобы не писать модульные тесты, например: *изначально не было речи о фреймворке для тестирования или из-за этого время разработки увеличится на 20 %, или тесты устарели, поэтому мы больше их не запускаем*. Правда состоит в том, что в наши дни мы не может позволить себе не писать тесты. Приложения увеличиваются в размерах и становятся более сложными, и требования со временем меняются. Приложение с хорошим набором тестов можно изменить гораздо быстрее, и оно будет гораздо более устойчивым к будущим изменениям требований, чем приложение, у которого нет тестов. Вот когда реальная экономия затрат на модульное тестирование становится очевидной. Написав модульные тесты для своего приложения, вы проверяете его будущее и гарантируете, что любое изменение в кодовой базе не нарушит существующую функциональность.

Мы также хотим писать свои приложения, чтобы они выдержали испытание временем. Код, который мы сейчас пишем, может годами находиться в производственной среде, а это значит, что иногда вам придется вносить улучшения или исправлять ошибки в коде, который был написан много лет назад. Если приложение имеет полный набор окружающих его тестов, тогда внесение изменений может быть выполнено с уверенностью, что эти изменения не нарушат существующую функциональность.

Разработка через тестирование в пространстве JavaScript также добавляет еще один слой к нашему покрытию кода. Довольно часто команды разработчиков пишут тесты, предназначенные только для логики приложения на стороне сервера.

Например, в пространстве Visual Studio эти тесты часто пишутся только с ориентацией на MVC-фреймворк, состоящий из контроллеров, представлений и базовой бизнес-логики. Всегда было довольно сложно проверить логику приложения на стороне клиента, другими словами, фактический визуализированный HTML-код и пользовательские взаимодействия.

Фреймворки для тестирования JavaScript-кода предоставляют нам инструменты для устранения этого пробела. Теперь мы можем приступить к модульному тестированию своего визуализированного HTML-кода, а также моделировать взаимодействия с пользователем, такие как заполнение форм и нажатие кнопок. Этот дополнительный уровень тестирования в сочетании с тестированием на стороне сервера означает, что у нас есть способ модульного тестирования каждого уровня нашего приложения – от бизнес-логики на стороне сервера до рендеринга страниц на стороне сервера и вплоть до взаимодействия с пользователем. Эта возможность выполнять модульное тестирование взаимодействий пользователя с клиентской частью интерфейса является одной из самых сильных сторон любого MV*-фреймворка JavaScript. Фактически это может даже повлиять на архитектурные решения, которые вы принимаете при выборе технологического стека.

Модульные, интеграционные и приемочные тесты

Автоматизированные тесты можно разбить на три основные области, или типы тестов: модульные тесты, интеграционные тесты и приемочные тесты. Также можно описать эти тесты как тесты из «черного ящика» или из «белого ящика». Тесты из белого ящика – это тесты, в которых внутренняя логика или структура тестируемого кода известна тестировщику. Тесты из черного ящика, с другой стороны, – это тесты, в которых внутренний дизайн и/или логика тестировщику неизвестны.

Модульные тесты

Модульный тест – это, как правило, тест из белого ящика, в котором все внешние интерфейсы блока кода мокированы или заглушены. Если мы тестируем некоторый код, который, например, выполняет асинхронный вызов, чтобы загрузить блок JSON, для модульного тестирования этого кода потребуется мокирование возвращенного JSON. Этот метод гарантирует, что тестируемому объекту всегда дается известный набор данных. Когда появляются новые требования, этот известный набор данных может расти и расширяться, конечно. Тестируемые объекты должны быть спроектированы для взаимодействия с интерфейсами, чтобы эти интерфейсы можно было легко мокировать или заглушить в сценарий модульного тестирования.

Интеграционные тесты

Интеграционные тесты – это еще одна форма тестов из белого ящика, которые позволяют тестируемому объекту работать в среде, близкой к реальному коду. В нашем предыдущем примере, когда некоторый код выполняет асинхронный вызов для загрузки блока JSON, интеграционному тесту на самом деле нужно вызвать службы REST, которые генерируют JSON. Если эта служба REST полагается на данные из базы данных, тогда для интеграционного теста потребуются данные в базе данных, которые соответствуют сценарию интеграционного теста. Если бы мы описали модульный тест как имеющий границу вокруг тестируемого объекта, то интеграционный тест – это просто расширение этой границы для включения зависимых объектов или служб.

Создание автоматизированных интеграционных тестов для ваших приложений значительно улучшит качество вашего продукта. Рассмотрим случай сценария, который мы использовали, когда блок кода вызывает службу REST для данных в формате JSON. Кто-то может легко изменить структуру этих данных, возвращаемых службой REST. Наши модульные тесты по-прежнему будут проходить, так

как на самом деле они не вызывают REST-код на стороне сервера, но наше приложение будет испорчено, потому что возвращенный JSON не соответствует нашим ожиданиям.

Без интеграционных тестов эти типы ошибок будут обнаруживаться только на более поздних этапах ручного тестирования. Думая об интеграционных тестах, реализуя конкретные наборы данных для интеграционных тестов и встраивая их в свой набор, вы сможете быстро устранить подобные ошибки.

Приемочные тесты

Приемочные тесты – это тесты из черного ящика, и, как правило, они основаны на сценариях. Они могут включать в себя несколько пользовательских экранов или пользовательских взаимодействий для прохождения. Эти тесты также обычно выполняются группой тестирования, так как могут потребовать входа в приложение, поиска определенного набора данных, обновления данных и т. д. Используя планирование и множество доступных инструментов, мы также можем автоматизировать эти приемочные тесты, чтобы они запускались как часть автоматизированного набора тестов. Чем больше приемочных тестов будет у проекта, тем надежнее он будет.

Обратите внимание, что в методологии разработки через тестирование каждая ошибка, обнаруженная группой ручного тестирования, должна приводить к созданию новых модульных, интеграционных или приемочных тестов. Эта методология поможет гарантировать, что как только ошибка будет найдена и исправлена, она больше не появится.

Фреймворки для модульного тестирования

Есть множество JavaScript-фреймворков для модульного тестирования, а также несколько фреймворков, написанных на TypeScript. Два самых популярных – это Jasmine (`http://jasmine.github.io/`) и QUnit (`http://qunitjs.com/`). Если вы пишете код TypeScript на основе Node, можете взглянуть на Mocha (`https://github.com/mochajs/mocha/wiki`).

Хотя и были попытки написания фреймворков для модульного тестирования на TypeScript и для TypeScript, таких как MaxUnit от KnowledgeLake или tsUnit от Steve-Fenton, эти фреймворки так и не были докончены. Их ограниченный набор функций, по сравнению с закаленными в боях, испытанными и проверенными фреймворками, означал, что они должны были наверстать упущенное, и в конечном итоге они были заброшены.

Простота интеграции TypeScript с библиотеками JavaScript означает, что разработчики, ищущие полнофункциональный набор инструментов для модульного тестирования, могут повторно использовать библиотеки для тестирования JavaScript-кода, как если бы они были написаны на TypeScript.

В оставшейся части этой главы мы будем использовать Jasmine 3.2 в качестве фреймворка для тестирования.

Jasmine

Jasmine – это основанный на поведении фреймворк для тестирования JavaScript-кода, который существует дольше, чем некоторые из старейших фреймворков JavaScript. Backbone впервые появился в октябре 2010 года, а Jasmine был выпущен на месяц раньше, в сентябре 2010 года. Первоначально Jasmine был портом популярного фреймворка для модульного тестирования программного обеспечения на языке Java, jUnit и назывался jsUnit. Возраст этого фреймворка показывает, что он выдержал испытание временем и постоянно пополняется новыми функциями, позволяющими использовать его даже самым современным фреймворкам.

У Jasmine очень простой синтаксис, и он разработан так, чтобы его можно было легко прочитать и понять. Он также может быть легко расширен и является рекомендуемой средой для Aurelia, а также модульного и интеграционного тестирования в Angular. Установка Jasmine с использованием npm выглядит так:

```
npm install jasmine --save
```

Соответствующие файлы объявлений Jasmine можно установить с помощью @types:

```
npm install @types/jasmine -save-dev
```

Мы можем запустить Jasmine непосредственно из командной строки в Node, установив Jasmine как глобальный модуль:

```
npm install -g jasmine
```

После установки нам нужно будет инициализировать наш каталог проекта, который будет использовать Jasmine:

```
jasmine init
```

Будет создана директория spec, где должны находиться наши тесты и файл `spec/support/jasmine.json`:

```
{
  "spec_dir": "spec",
  "spec_files": [
```

```
      "**/*[sS]pec.js"
    ],
    "helpers": [
      "helpers/**/*.js"
    ],
    "stopSpecOnExpectationFailure": false,
    "random": true
}
```

Jasmine настроил образец файла `jasmine.json`, у которого есть несколько стандартных свойств.

Свойство `spec_dir` сообщает Jasmine, что он должен искать в каталоге `/spec`, чтобы найти работающие тесты. Свойство `spec_files` указывает, что любой файл, заканчивающийся на `.spec.js` или `.Spec.js`, будет считаться тестовой спецификацией Jasmine. Мы пока не будем использовать каталог `helpers`, хотя он предназначен для загрузки файлов, которые могут понадобиться нашим тестам.

Свойство `stopSpecOnExpectationFailure`, значение которого установлено как `false`, означает, что Jasmine продолжит выполнять все тесты в наборе, независимо от сбоя. Свойство `random` используется для случайного выбора теста для запуска из всего набора.

Наш полный набор тестов (которые на самом деле еще не тестировались) можно запустить следующим образом:

```
jasmine
```

Поскольку в папке specs пока нет тестов, Jasmine сообщит об этом:

```
Finished in 0.001 seconds
Incomplete: No specs found
```

Простой тест

Jasmine использует простой формат для написания тестов. Давайте создадим файл `spec/SimpleJasmine.spec.ts` со следующим кодом TypeScript:

```
describe('spec/SimpleJasmine.spec.ts', () => {
  it('should fail', () => {
    let undefinedValue;
    expect(undefinedValue).toBeDefined('should be defined');
  })
});
```

Этот фрагмент начинается с вызова функции describe, которая принимает два аргумента.

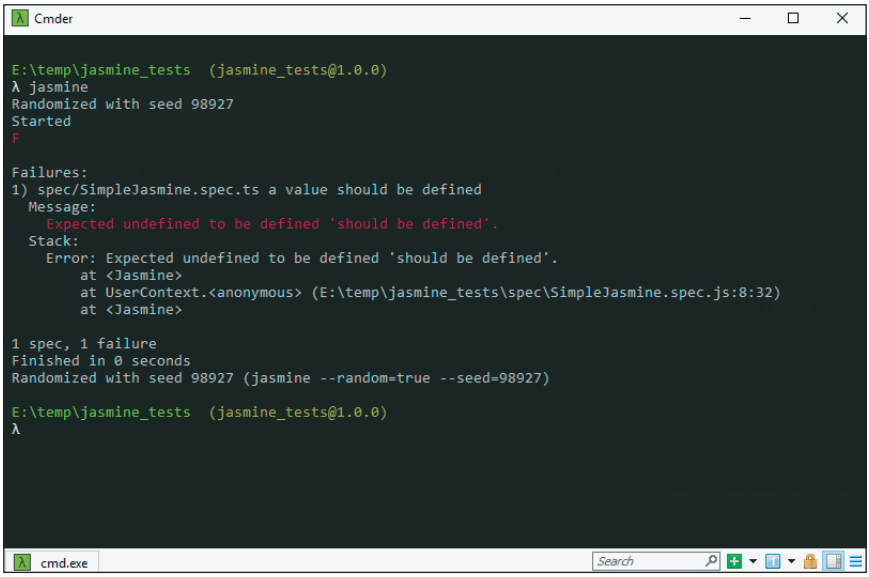
Первый аргумент – это имя набора тестов, а второй – анонимная функция, которая содержит каждый тест в нашем наборе тестов. В рамках этой анонимной функции мы вызываем функцию `it`, чтобы описать реальный тест, который также принимает два аргумента. Первый аргумент – это имя теста, а второй – анонимная функция, которая содержит наш фактический тест.

Этот тест начинается с определения переменной `undefinedValue`, но фактически не устанавливает ее значение. Далее мы используем функцию `expect`. Просто прочитав код этого оператора, мы сможем быстро понять, что делает модульный тест. Ожидается, что значение `undefinedValue` должно быть определено, то есть не должно быть равно `undefined`.

Функция `expect` принимает один аргумент и возвращает значение, которое можно использовать в свободном синтаксисе для оценки так называемым сопоставителем. Сопоставитель в этом случае – `toBeDefined`. Таким образом, функция `expect` передает значение, которое было присвоено сопоставителю, которое либо пройдет, либо потерпит неудачу. Большинство сопоставителей может быть вызвано с одной строкой в качестве аргумента, которая будет просто записывать переданное сообщение в консоль. Ключевое слово `expect` аналогично ключевому слову `Assert` в других библиотеках тестирования.

В этом случае наш сопоставитель ожидает, что значение переменной `undefinedValue` должно быть определено, поэтому данный тест должен немедленно завершиться неудачей.

Мы можем запустить простой тест из командной строки, который потерпит неудачу, выдав следующее сообщение:



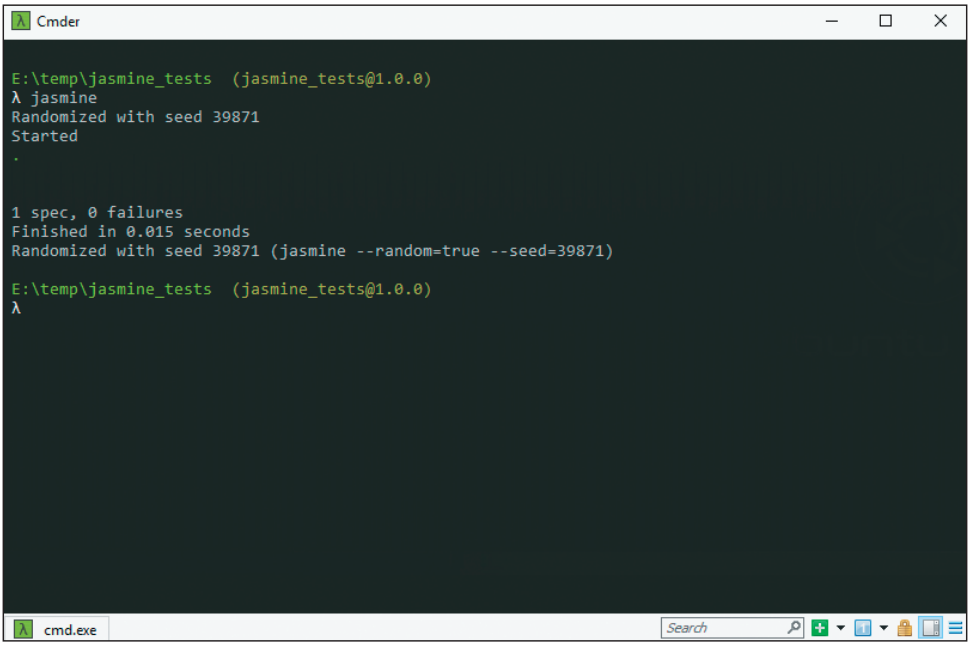
Здесь видно, что наш тест не пройден и сообщение 'should be defined'. Jasmine также сообщает нам, где именно в нашем JavaScript-файле тест не пройден, что очень поможет при отладке.

Обратите внимание на название, которое мы даем этому набору тестов. Наша функция describe вызывается с именем фактического файла TypeScript на диске, `spec/SimpleJasmine.spec.ts`. Хотя название набора тестов может быть любым, при запуске сотен тестов оно помогает точно определить, какой файл был ответственным за неудачный тест. Когда тест не будет пройден, Jasmine запишет трассировку полного стека для сбоя, которая может легко составить более 20 строк трассировки стека в производственных системах. Возможность незамедлительного отслеживания причины сбоя в конкретном тестовом файле очень помогает при поиске причины сбоя.

Теперь, когда у нас есть тест, который терпит неудачу, мы можем выполнить успешное прохождение теста. Это так же просто, как присвоить значение:

```
let undefinedValue = "test";
expect(undefinedValue).toBeDefined('should be defined');
```

Здесь мы присвоили значение "test" переменной undefinedValue. Этот тест теперь пройдет следующим образом:



Отлично. Мы следуем мантре TDD, сначала написав провальный тест и запустив его, чтобы убедиться, что он не пройден, а после напишем код для успешного прохождения теста.

Репортеры

Если мы посмотрим на вывод консоли библиотеки модульного тестирования Jasmine, то увидим, что она показывает зеленую точку для каждого теста, который мы запустили, а затем количество выполненных тестов и сбоев, с которыми мы столкнулись. Но что, если мы хотели бы видеть имя каждого теста, который мы запустили на консоли? Вот тут вступают в действие репортеры Jasmine. Репортеры Jasmine позволяют изменять выходные данные каждого набора тестов и каждого теста, чтобы дать более подробное описание. Мы можем настроить тестового репортера, вставив некий код в файл в каталоге `helpers`, который Jasmine будет загружать и запускать перед каждым тестовым запуском.

Несколько репортеров Jasmine уже написано и включено в пакет npm под названием `jasmine-reporters`. Нам нужно будет установить его:

```
npm install jasmine-reporters --save-dev
```

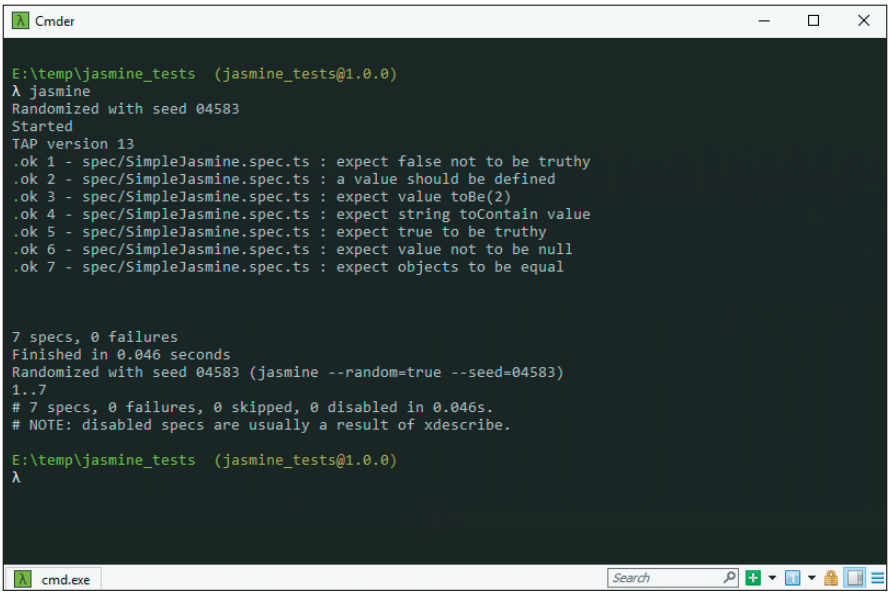
Давайте создадим файл конфигурации JavaScript в каталоге `spec/helpers` с именем `attachTapReporter.js`:

```
var reporters = require('jasmine-reporters');
var tapReporter = new reporters.TapReporter();
jasmine.getEnv().addReporter(tapReporter)
```

Здесь мы просто загружаем пакет `jasmine-reporters` посредством вызова функции `require`. Это относится к возможности загрузки модулей в JavaScript, которую мы рассмотрим в следующей главе.

Один из объектов, который мы можем создать из пакета `jasmine-reporters`, называется `TapReporter`. Мы создаем новый экземпляр этого класса, а затем получаем дескриптор среды Jasmine через вызов `jasmine.getEnv()`. После получения дескриптора мы можем вызвать функцию `addReporter` и передать экземпляр нашего класса `TapReporter`.

Если мы теперь запустим наши тесты из командной строки, то увидим более подробное описание каждого теста, записываемое в консоль:



Обратите внимание, что вывод, который мы видим, является результатом выполнения семи тестов, каждый с именем набора тестов и именем самого теста. Этот скриншот фактически включает в себя все тесты, которые мы напишем в следующем разделе.

Сопоставители

Как было показано в первом простом тесте, Jasmine использует свободный синтаксис, позволяющий присоединять сопоставители Jasmine после оператора `expect(...)`. В первом тесте мы использовали сопоставитель `.toBeDefined`. Jasmine, однако, обладает широким спектром сопоставителей, которые можно использовать в тестах, а также позволяет нам писать и включать настраиваемые сопоставители. Давайте кратко рассмотрим некоторые из них:

```
it("expect value toBe(2)", () => {
   let twoValue = 2;
   expect(twoValue).toBe(2);
})
```

Здесь мы используем сопоставитель `.toBe`, чтобы проверить, что значение переменной `twoValue` действительно равно 2.

```
it("expect string toContain value ", () => {
   let testString = "12345a";
   expect(testString).toContain("a");
});
```

В этом тесте мы используем сопоставитель `toContain`, чтобы проверить, что строка `"12345a"` содержит значение `"a"`.

```
it("expect true to be truthy", () => {
   let trueValue = true;
   expect(trueValue).toBeTruthy();
});
```

В этом тесте мы используем сопоставитель `toBeTruthy`, чтобы проверить, что для переменной `trueValue` установлено логическое значение `true`.

Мы также можем изменить значение любого оператора ожидания, используя сопоставитель `.not`:

```
it("expect false not to be truthy", () => {
   let falseValue = false;
   expect(falseValue).not.toBeTruthy();
});
```

Здесь мы используем сопоставитель `.not`, а затем `toBeTruthy`, чтобы проверить, что переменная `falseValue` действительно ложна. Мы также можем использовать `.not` в других комбинациях:

```
it("expect value not to be null", () => {
   let definedValue = 2;
   expect(definedValue).not.toBeNull();
});
```

Этот тест проверяет, что значение переменной `defineValue` не равно нулю, с помощью сопоставителя `toBeNull`.

Мы также можем проверить, что два объекта JavaScript равны:

```
it("expect objects to be equal", () => {
   let obj1 = {a : 1, b : 2};
   let obj2 = {b : 2, a : 1};
   expect(obj1).toEqual(obj2);
});
```

В этом тесте мы определили два объекта с именами `obj1` и `obj2`, которые имеют одинаковые свойства. Средство сопоставления `toEqual` правильно определит, что эти два объекта имеют одинаковые свойства и значения и поэтому считаются равными.

Обязательно посетите сайт Jasmine, чтобы ознакомиться с полным списком сопоставителей, а также подробностями написания пользовательских сопоставителей.

Запуск и завершение теста

Как и в других фреймворках для тестирования, Jasmine предоставляет механизм для определения функций, которые будут выполняться до и после каждого теста, или как механизм запуска и завершения теста. В Jasmine функции `beforeEach` и `afterEach` действуют так, как видно из приведенного ниже теста:

```
describe("beforeEach and afterEach tests", () => {

   let myString: string | undefined;

   beforeEach(() => {
      myString = "this is a string";
   });
   afterEach(() => {
      expect(myString).toBeUndefined();
   });

   it("should find then clear the myString variable", () => {
      expect(myString).toEqual("this is a string");
      myString = undefined;
   });
});
```

В этом тесте вначале мы определяем переменную с именем `myString`. Как мы знаем из правил лексической области видимости JavaScript, эта переменная будет доступна для использования в области видимости вложенной функции, т. е. функции `describe`. Это означает, что переменная `myString` будет доступна в каждой из следующих функций: `beforeEach`, `afterEach` и `it`. В нашей функции `beforeEach` эта переменная имеет строковое значение `"this is a string"`. В функции `afterEach` переменная проверяется, чтобы увидеть, что она была переустановлена в значение `undefined`. В рамках наших проверок мы ожидаем, что эта переменная была установлена с помощью функции `beforeEach`. В конце теста мы переустанавливаем переменную в значение `undefined`. Обратите внимание, что функция `afterEach` также вызывает `expect` – в этом случае, чтобы гарантировать, что тест вернул переменную обратно в неопределенное `undefined`.

В том же духе Jasmine предоставляет функции `beforeAll` и `afterAll`, которые будут вызываться до запуска полного набора тестов и после запуска. Эти функции обычно используются для настройки экземпляров классов или переменных, которые понадобятся каждому тесту. Типичным примером этого может быть, например, настройка подключения к базе данных. Первоначальное создание соединения с базой данных обычно занимает много времени, поэтому его можно настроить в функции `beforeAll`, а затем закрыть в функции `afterAll`.

Принудительные тесты

Когда ваш набор тестов начинает расти, во время разработки становится необходимым ограничить запуск всего набора одним конкретным тестом или одним конкретным набором тестов. Обычно это делается для того, чтобы найти причину определенного сбоя или сосредоточиться на одном наборе тестов во время разработки кода. Jasmine предоставляет функции `fdescribe` и `fit` для принудительного выполнения тестов:

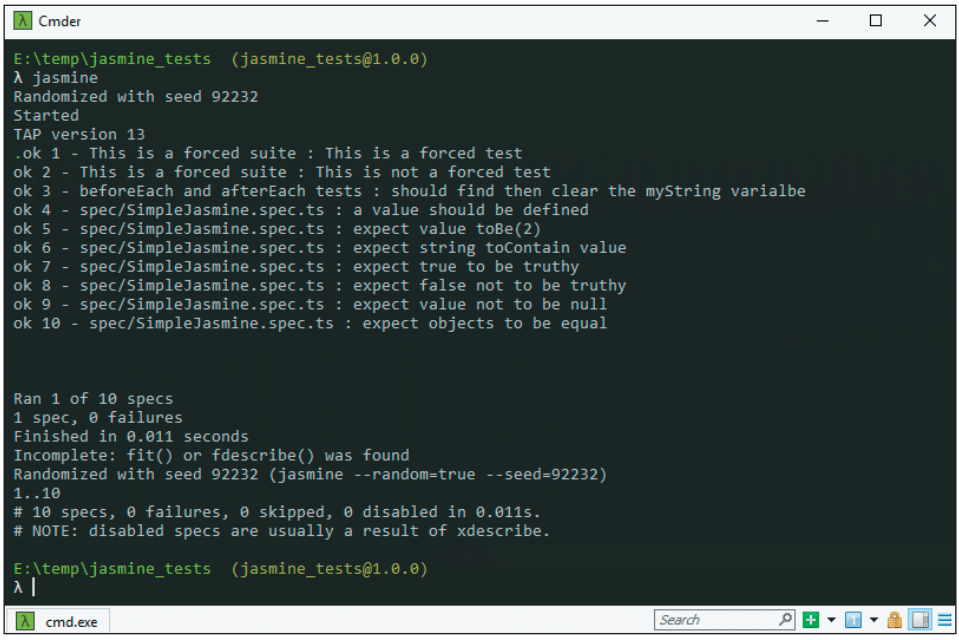
```
fdescribe("This is a forced suite", () => {
   it("This is not a forced test", () => {
      expect(true).toBeFalsy('true should be false');
   });
   fit("This is a forced test", () => {
      expect(false).toBeFalsy();
   })
});
```

Здесь мы заменили функцию `describe` функцией `fdescribe`. Или, говоря проще, мы вставили перед `describe` букву `f`. Это заставит Jasmine принудительно запустить этот тест за счет любых других тестов. Другими словами, Jasmine не будет запускать другие наборы тестов, кроме этого. Это очень удобно во время разработки, так как мы можем ограничить запуск теста конкретным набором при разработке тестов.

Обратите также внимание на то, что мы заменили функцию `it` функцией `fit` во втором тесте этого набора. Поставив букву `f` перед функцией `it`, мы можем принудительно запустить только определенный тест. В этом примере первый тест не будет пройден, так как он ожидает `true toBeFalsy`. Второй тест, однако, будет успешным. Если бы мы запустили наш набор тестов на этом этапе, мы бы обнаружили, что одна спецификация была запущена с нулевыми ошибками.

Функции `fdescribe` и `fit` работают с различными типами ограничений от самых высоких до наименее высоких. Другими словами, если ни один из тестов не был помечен как `fit`, тогда будут выполнены все тесты в наборе `fdescribe`. Если несколько тестов в наборе `fdescribe` были помечены как `fit`, то будут выполняться только те, которые отмечены как `fit`. Если набор не был отмечен как `fdescribe`, но один тест был помечен как `fit`, тогда будет выполняться только этот тест.

Обратите внимание, что ни при каких обстоятельствах вы не должны фиксировать изменения в тестах, помеченных как `fdescribe` или `fit`. Это будет означать, что ваши серверы сборки будут выполнять только небольшую часть тестов, а не весь набор. Jasmine на самом деле предупредит вас, если найдет какие-либо тесты, которые были принудительными, и запишет это в консоль, как видно на скриншоте ниже:



Здесь видно, что была запущена одна спецификация из десяти и что Jasmine пометил тест как незавершенный.

Пропуск тестов

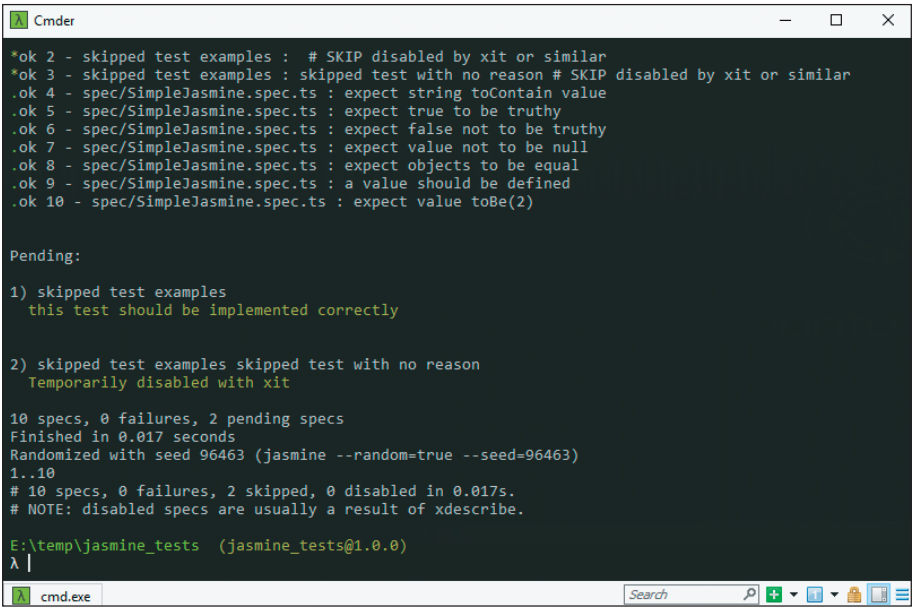
Аналогично принудительному запуску тестов, тесты могут быть пропущены с использованием `xit` вместо `it` и `xdescribe` вместо `describe`. Это означает, что тесты не будут выполняться как часть вашего набора. Хотя существуют веские причины пропускать тесты в производственной системе, эти причины всегда должны быть чрезвычайным обстоятельством и всегда должны быть очень недолгими.

Можно пропустить тест двумя способами. Первый – пометить тест буквой `x`, поэтому вместо `it` используйте `xit`. Второй способ пропустить тест – вызвать функцию `pending`:

```
describe("skipped test examples", () => {
   xit("skipped test with no reason", () => {
      expect(false).toBeTruthy();
   });
   it("", () => {
      expect(false).toBeTruthy();
      pending("this test should be implemented correctly");
   })
});
```

Здесь у нас два теста. Первый тест пропускается с помощью `xit`, а второй – из-за вызова функции `pending`. Обратите внимание, что вызов функции `pending` на самом деле идет после оператора ожидания. Если мы внимательно посмотрим на оператора ожидания, то увидим, что этот тест должен потерпеть неудачу, поскольку мы ожидаем `false toBeTruthy`. Это означает, что вызов функции `pending` может происходить в любом месте самого теста, и когда Jasmine находит вызов `pending`, он пропускает весь тест.

Существует тонкое и важное различие между использованием `xit` и функции `pending`. Функция `pending` указывает причину пропуска теста. Если мы запустим этот тест, то увидим следующее:



Здесь видно, что два теста были пропущены. Первый тест показывает сообщение "`Temporarily disabled with xit`", в то время как второй тест показывает сообщение, которое мы указали в вызове функции pending: "`this test should be implemented correctly`". Это очень ясно показывает всем членам нашей команды, почему данный тест был пропущен. Он действует как сигнал о том, что тест должен быть повторно включен, если причина для того, чтобы пропустить его, больше не действительна.

Тесты, управляемые данными

Бывают моменты, когда набор тестов для одного и того же фрагмента кода нужно повторять снова и снова с несколько разными входными данными. Например, если вы тестировали способность функции распознавать вхождение строки, вам

нужно выполнить этот тест с несколькими различными строками. Такие тесты называются тестами на основе данных. Результат каждого теста будет одинаковым, но мы должны убедиться, что тестируем свой код в различных обстоятельствах.

Чтобы показать, насколько расширяемой является библиотека Jasmine, *Ж. П. Кастро* написал очень короткую, но мощную утилиту для проведения тестов, управляемых данными в рамках Jasmine. Его блог на эту тему можно найти по адресу `http://blog.jphpsf.com/2012/08/30/diring-up-your-javascript-jasmine-tests`, и хранилище GitHub можно найти на странице `https://github.com/jphpsf/jasmine-data-provider`. Это простое расширение позволяет писать интуитивно понятные тесты Jasmine, которые принимают параметр в качестве части каждого теста:

```
describe("data driven tests", () => {
   using("valid values", [
      "first string",
      "second_string",
      "!!string!!"
   ], (value) => {
      it(`${value} should contain 'string'`, () => {
         expect(value).toContain("string");
      });
   });
});
```

Обратите внимание на использование функции `using` внутри функции `describe`. Эта функция позволяет вызывать наш тест несколько раз, при этом каждый вызов теста использует последующее значение нашего массива. Функция `using` принимает три параметра – строковое описание набора значений, массив значений и определение функции – и затем вызывает сам тест.

Таким образом, наш тест в этом примере будет вызван три раза, первый раз со значением `"first string"`, второй раз со значением `"second_string"` и третий раз со значением `"!! string !!"`. Обратите также внимание на то, что в вызове `it` мы меняем имя теста на лету, чтобы включить в него передаваемый параметр `value`. Это необходимо для того, чтобы у каждого теста было уникальное имя.

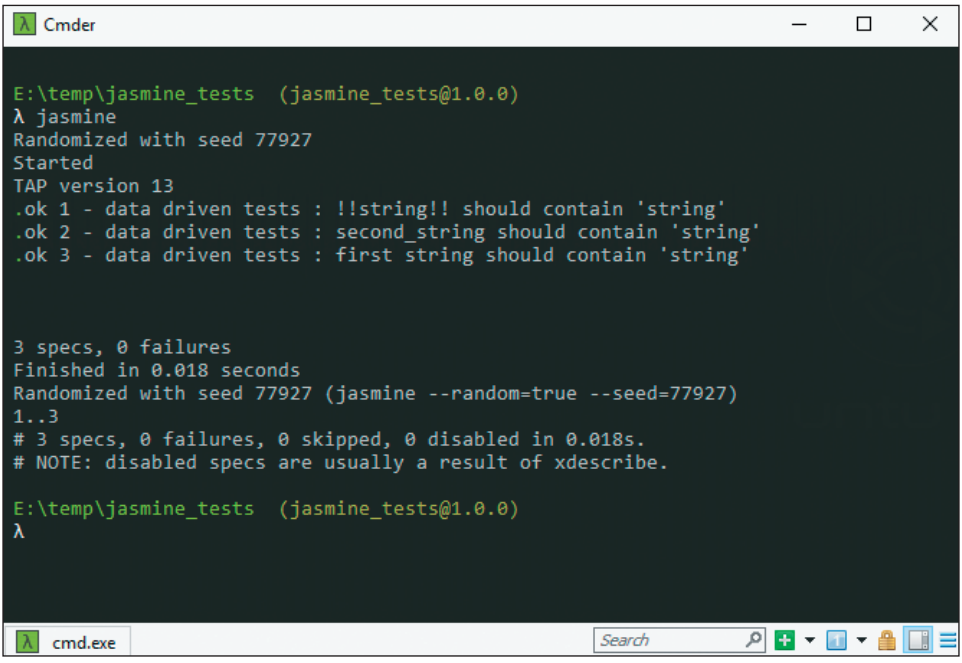
Реализация этого использования функции в блоге Ж. П. Кастро была написана на языке JavaScript, но мы можем легко воспроизвести эту простую функцию в TypeScript:

```
export function using<T>
   (name: string, values: T[], func: Function) {
   for (var i = 0, count = values.length; i < count; i++) {
      func.apply(Object, [values[i]]);
   }
}
```

Здесь мы используем синтаксис обобщений для определения функции `using`, которая объявлена с типом `T`. У нее три параметра. Первый параметр – это строка, которая является именем набора данных. Второй параметр – это массив значений типа `T`. Использование синтаксиса обобщений для этого параметра означает, что мы ограничили все элементы в массиве одним типом. Это означает, что мы, например, не можем смешивать числа и строки в нашем массиве. Третий параметр – это функция, которая будет содержать сам тест.

В рамках данной функции мы просто перебираем все элементы в массиве и вызываем функцию `apply` для функции обратного вызова, которая была передана как аргумент `func`. Функция `apply` вызовет функцию с заданным значением `this` и аргументами функции, указанными как массив. Обратите внимание, что мы используем `Object` в качестве первого аргумента функции `apply`. Это означает, что исходное значение `this` будет использовано при вызове самой функции. По сути, это вызовет любую функцию, которую мы передали (что является нашим тестом) несколько раз, по одному разу для каждого значения в массиве и с правильным значением `this`.

Итак, с помощью небольшого креативного кода мы реализовали для Jasmine возможность вызывать тесты с массивом значений, управляемых данными. Результат этого теста выглядит так:



Здесь видно, что наш тест был запущен три раза, по одному для каждого из значений, которые мы использовали в нашем массиве.

Использование шпионов

Jasmine также обладает очень мощным свойством, которое позволяет вашим тестам увидеть, была ли вызвана определенная функция, а также определить фактические параметры, с которыми она была вызвана. Это свойство известно как слежка за функцией. Когда мы создаем шпиона, то временно угоняем вызов функции и переопределяем ее шпионской функцией Jasmine. Давайте посмотрим на простого шпиона:

```
class MySpiedClass {
   testFunction(arg1: string) {
      console.log(arg1);
   }
}

describe("simple spy", () => {
   it("should spyOn a function call", () => {
      let classInstance = new MySpiedClass();
      let testFunctionSpy
         = spyOn(classInstance, 'testFunction');
      classInstance.testFunction("test");
      expect(testFunctionSpy).toHaveBeenCalled();
   });
});
```

Вначале идет класс `MySpiedClass` с единственной функцией `testFunction`. Эта функция принимает один аргумент и записывает его в консоль.

Наш тест начинается с создания нового экземпляра класса `MySpiedClass`, который присваивается переменной `classInstance`. Затем мы создаем шпион Jasmine с именем `testFunctionSpy`, вызывая функцию `spyOn`. Эта функция принимает два аргумента – сам экземпляр класса и имя функции, за которой будет осуществляться слежка. В этом тесте экземпляр класса называется `classInstance`, а функция, за которой мы хотим следить, носит название `testFunction`.

После того как мы создали шпиона, мы можем вызвать функцию и установить ожидание того, была ли вызвана функция. Это сущность шпиона. Jasmine будет наблюдать за функцией `testFunction` экземпляра класса `MySpiedClass`, чтобы узнать, был ли она вызвана.

Шпионы Jasmine по умолчанию блокируют вызов базовой функции. Другими словами, они заменяют функцию, которую вы пытаетесь вызвать, делегатом. Это часть процесса перехвата, который мы упомянули ранее. Если вам нужно шпионить за функцией, но также нужно тело, чтобы выполнить эту функцию, вы должны указать это поведение, используя свободный синтаксис `.and.callThrough()`.

Хотя это очень тривиальный пример, шпионы становятся достаточно влиятельными в различных сценариях тестирования. Мы можем вызвать метод класса и проверить, что каждый вызов функции, который делает этот класс, выполняется правильно. Шпионов также можно использовать для возврата данных, что очень удобно при использовании классов, которые вызывают конечные точки службы REST. Когда мы запускаем модульный тест, мы можем смоделировать фактический вызов конечной точки REST, чтобы она никогда не вызывалась во время выполнения теста.

Слежка за функциями обратного вызова

Давайте посмотрим, как можно использовать шпиона для проверки правильности вызова функции обратного вызова. Рассмотрим следующий код TypeScript:

```
class CallbackClass {
   doCallback(id: number, callback: (result: string) => void ) {
      let callbackValue = "id:" + id.toString();
      callback(callbackValue);
   }
}
class DoCallback {
   logValue(value: string) {
      console.log(value);
   }
}
```

Сперва мы определяем класс с именем `CallbackClass`, который имеет единственную функцию `doCallback`. Эта функция принимает аргумент `id` типа `number`, а также функцию `callback`. Функция `callback` должна принимать строку в качестве аргумента и возвращать `void`.

Второй класс, который мы определили, – это `DoCallBack` с единственной функцией `logValue`. Сигнатура этой функции соответствует сигнатуре функции обратного вызова, необходимой для функции `doCallback`, которую мы определили ранее. Используя шпионов Jasmine, мы теперь можем проверить логику функции `doCallback`.

Эта функция должна создать строку на основе переданного аргумента `id`, а затем вызвать функцию `callback`. Поэтому наши тесты должны выполнить две вещи. Во-первых, нам нужно убедиться, что строка, сгенерированная в функции `doCallback`, отформатирована правильно, а во-вторых, нам нужно убедиться, что наша функция обратного вызова действительно была вызвана с правильными параметрами. Наш тест Jasmine в этом случае будет выглядеть так:

```
describe("using callback spies", () => {
   it("should execute callback with the correct string value",
```

```
      () => {
      let doCallback = new DoCallBack();
      let classUnderTest = new CallbackClass();
      let callbackSpy = spyOn(doCallback, 'logValue');
      lassUnderTest.doCallBack(1, doCallback.logValue);

      expect(callbackSpy).toHaveBeenCalled();
      expect(callbackSpy).toHaveBeenCalledWith("id:1");
   });
});
```

Сначала создается экземпляр класса `CallbackClass`, а также экземпляр класса `DoCallback`. Затем мы создаем шпиона для функции `logValue` класса `DoCallback`. Помните, что функция `logValue` передается в функцию `doCallBack` в качестве параметра функции обратного вызова и вызывается с форматированной строкой.

Наши операторы `expect` в последних двух строках подтверждают, что эта цепочка обратных вызовов действительно выполнена правильно. Первый оператор просто проверяет, что была вызвана функция `logValue`, а второй проверяет, что она была вызвана с правильными параметрами. Итак, мы протестировали внутреннюю реализацию `CallbackClass` и функции `doCallback`, проверили, правильно ли она форматирует строку, а также проверили, что была вызвана сама функция обратного вызова.

Использование шпионов в качестве фальшивок

Еще одним преимуществом шпионов Jasmine является то, что они могут действовать как фальшивки. Другими словами, вместо вызова реальной функции вызов временно переопределяется, чтобы вызвать функцию-фальшивку. Функции-фальшивки также могут возвращать значения, что может быть очень полезно при генерации небольших фреймворков с поддержкой мокирования. Рассмотрим следующий тест:

```
class ClassToFake {
   getValue(): number {
      return 2;
   }
}
describe("using fakes", () => {
   it("calls fake instead of real function", () => {
      let classToFake = new ClassToFake();
      spyOn(classToFake, 'getValue').and.callFake ( () => {
```

```
        return 5;
      });
      expect(classToFake.getValue()).toBe(5);
    });
  });
```

Первым идет класс `ClassToFake` с единственной функцией `getValue`, которая возвращает 2. Затем наш тест создает экземпляр этого класса. Далее мы вызываем функцию `spyOn` для создания шпиона для функции `getValue`, а после используем синтаксис `.and.callFake` для присоединения анонимной функции в качестве функции-фальшивки. Эта функция вернет 5 вместо исходной функции `getValue`, которая должна была бы вернуть 2. Затем тест проверяет, вернет ли значение 5 вызов функции `getValue` для экземпляра класса `ClassToFake`. В этом тесте Jasmine заменит нашу новую функцию-фальшивку на исходную функцию `getValue` и, следовательно, вернет 5 вместо 2.

В Jasmine существует несколько вариантов синтаксиса функций-фальшивок, включая методы выдачи ошибок или возврата значений – опять же, обратитесь к документации по Jasmine для получения полного списка возможностей этих функций.

Асинхронное тестирование

Асинхронная природа JavaScript, ставшая популярной благодаря AJAX и jQuery, всегда была одной из основных составляющих этого языка и является основной архитектурой для приложений на базе Node. Давайте кратко рассмотрим асинхронный класс, а затем опишем, как следует его тестировать. Рассмотрим следующий код TypeScript:

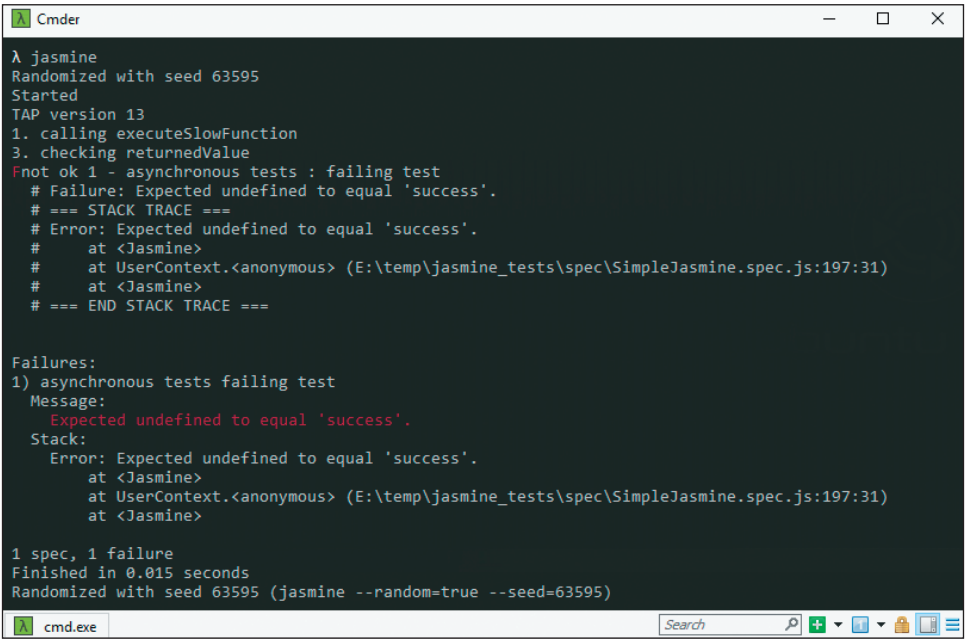
```
class MockAsyncClass {
   executeSlowFunction(success: (value: string) => void) {
      setTimeout(() => {
      success("success");
      }, 1000);
   }
}
```

В классе `MockAsyncClass` есть единственная функция `executeSlowFunction`, которая принимает функцию обратного вызова с именем success. В коде `executeSlowFunction` мы моделируем асинхронный вызов, используя функцию `setTimeout`, и вызываем успешную функцию обратного вызова только через `1000` миллисекунд (1 секунду). Поэтому эта функция имитирует асинхронную функцию, поскольку она будет выполнять обратный вызов только через целую секунду.

Наш тест функции executeSlowFunction может выглядеть так:

```
describe("asynchronous tests", () => {
   it("failing test", () => {
      let mockAsync = new MockAsyncClass();
      let returnedValue!: string;
      console.log(`1. calling executeSlowFunction`);
      mockAsync.executeSlowFunction((value: string) => {
         console.log(`2. executeSlowFunction returned`);
         returnedValue = value;
      });
      console.log(`3. checking returnedValue`);
      expect(returnedValue).toEqual("success");
   });
});
```

Сперва мы создаем экземпляр класса `MockAsyncClass`, а также переменную `returnValue`. Обратите внимание, что здесь нам нужно использовать определенный оператор присваивания (`!`), чтобы код правильно компилировался. Затем мы вызываем `executeSlowFunction`, используя анонимную функцию для параметра `success`. Эта анонимная функция устанавливает значение `returnedValue` для всего, что было передано из `MockAsyncClass`. Мы ожидаем, что значение `returnValue` должно равняться `success`, но если мы сейчас запустим этот тест, он завершится неудачно, выдав следующее сообщение об ошибке:



Поскольку `executeSlowFunction` является асинхронной, JavaScript не будет ждать, пока не будет вызвана функция обратного вызова, прежде чем выполнить следующую строку кода. Это можно проверить с помощью наших операторов записи в консоль. Ожидается, что поток выполнения нашего теста будет следовать 1, затем 2, а потом 3. Если мы проверим вывод консоли этого теста, то увидим следующее:

```
1. calling executeSlowFunction
3. checking returnedValue
```

Это означает, что оператор ожидания (в строке 3) вызывается до того, как `executeSlowFunction` сможет вызвать нашу анонимную функцию обратного вызова (установив значение `returnValue`). На самом деле весь тест был завершен до того, как у executeSlowFunction был какой-либо шанс записать что-либо в консоль.

Использование функции done()

Jasmine использует функцию `done`, чтобы помочь нам с такими асинхронными тестами. В функциях `beforeEach`, `afterEach` или `it` мы передаем аргумент с именем `done` (который является функцией), а затем вызываем его в конце нашего асинхронного кода. Давайте перепишем предыдущий тест `executeSlowFunction` следующим образом:

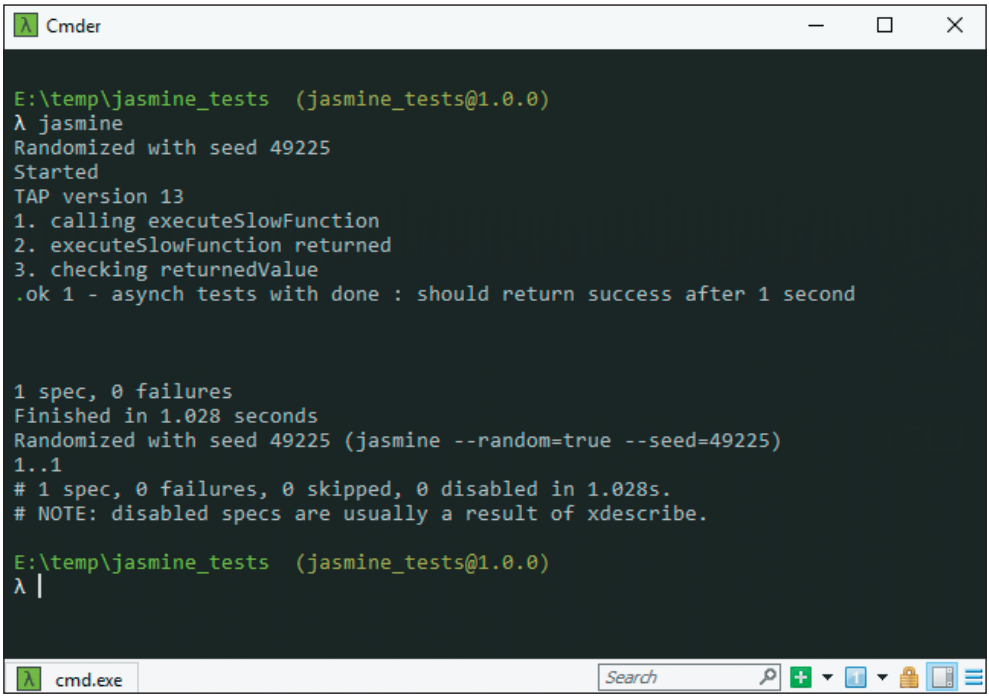
```
describe("asynch tests with done", () => {
     let returnedValue!: string;

   beforeEach((done) => {
      returnedValue = "no_return_value";
      let mockAsync = new MockAsyncClass();
      console.log(`1. calling executeSlowFunction`);
      mockAsync.executeSlowFunction((value: string) => {
         console.log(`2. executeSlowFunction returned`);
         returnedValue = value;
         done();
      });
   });

   it("should return success after 1 second", (done) => {
      console.log(`3. checking returnedValue`);
      expect(returnedValue).toEqual("success");
      done();
   });
});
```

В этой версии асинхронного теста мы переместили переменную `returnValue` за пределы нашего теста и включили функцию `beforeEach` для запуска до нашего реального теста.

Эта функция сначала сбрасывает значение `returnValue`, а затем устанавливает экземпляр класса `MockAsyncClass`. В конце она вызывает `executeSlowFunction` для этого экземпляра.

Обратите внимание, что функция `beforeEach` принимает параметр с именем `done`, а затем вызывает `done()` после вызова строки `returnValue = value`. Также обратите внимание, что второй параметр функции `it` теперь также принимает параметр `done` и вызывает `done()` после завершения теста.

Если мы выполним этот тест сейчас, то увидим следующее:



Здесь видно, что порядок журналов консоли действительно верен:

```
1. calling executeSlowFunction
2. executeSlowFunction returned
3. checking returnedValue
```

Итак, что мы сделали здесь? Мы изменили свой оригинальный тест и разбили его на две половины. Первая половина – это функция `beforeEach`, которая вызывает `executeSlowFunction`, сохраняя возвращаемое значение в перемен-

ной `returnValue`. Поэтому наш реальный тест ожидает выполнения функции `done()`, а затем запускает оставшуюся часть теста. Эта структура означает, что мы вызываем нашу асинхронную функцию и запускаем тест и операторы ожидания только после выполнения асинхронной функции.

Из документации к Jasmine: *«Спецификация не запустится, пока не будет вызвана функция* `done` *в вызове* `beforeEach`*, и эта спецификация не будет завершена, пока ее функция done не будет вызвана. По умолчанию Jasmine будет ждать пять секунд, прежде чем появится ошибка тайм-аута. Это можно изменить, используя переменную* `jasmine.DEFAULT_TIMEOUT_INTERVAL`*».*

Использование async awaitc

Если асинхронная функция, которую мы тестируем, использует промисы, мы легко можем включить синтаксис `async await` для запуска асинхронных тестов. В качестве примера давайте создадим класс, который использует промис для возврата значения:

```
class MockAsyncWithPromiseClass {
   delayedPromise(): Promise<string> {
      return new Promise<string>
      ((resolve: (str: string) => void,
         reject: (str: string) => void
      ) => {
         function afterTimeout() {
            console.log(`2. resolving promise`);
            resolve('success');
         }
         setTimeout(afterTimeout, 1000);
      });
   }
}
```

Здесь у нас есть класс `MockAsyncWithPromiseClass` с единственной функцией `delayedPromise`, которая возвращает промис типа `string`. В этой функции мы настроили анонимную функцию `afterTimeout`, которая выполнит промис со значением `success`. Затем мы вызываем эту внутреннюю анонимную функцию через 1 секунду. Как и в случае с нашими предыдущими асинхронными функциями, это означает, что промис будет выполнен только через 1 секунду. Наш модульный тест теперь может использовать ключевые слова `async` и `await`:

```
describe("async test with async keyword", () => {
   it("should wait for async to return with value ", async () => {
      let mockAsyncWithPromise = new MockAsyncWithPromiseClass();
      let returnedValue!: string;
      console.log(`1. calling delayedPromise`);
```

```
      returnedValue = await mockAsyncWithPromise.delayedPromise();
      console.log(`3. checking returnedValue`);
      expect(returnedValue).toEqual("success");
    });
});
```

Здесь у нас есть тест с именем "`it should wait for async to return`". Обратите внимание, что мы добавили ключевое слово async после описания теста. Это позволяет нам использовать ключевое слово await в этой тестовой функции.

Наш тест начинается с создания экземпляра класса `MockAsyncWithPromiseClass`. Затем мы вызываем функцию `delayedPromise`, используя ключевое слово `await`, и присваиваем результат переменной с именем `returnValue`. Обратите внимание, что у нас есть пара консольных журналов, чтобы показать порядок выполнения функций в рамках этого теста. Выполнение данного теста покажет, что использование ключевого слова `await` с функцией, которая возвращает промис, ведет себя, как ожидалось:

```
1. calling delayedPromise
2. resolving promise
3. checking returnedValue
```

HTML-тесты

Тесты, которые мы выполняли до этого момента, были довольно простыми, и для них не нужно иметь HTML-страницу или активный DOM. Как только мы начнем запускать тесты, требующие DOM, нам нужно будет внедрить наши тесты в работающий браузер, чтобы они могли работать правильно. Jasmine можно легко запустить в браузере, настроив HTML-страницу для запуска тестов. Как правило, этот файл будет называться `SpecRunner.html`:

```
<html>
<head>
  <link rel="stylesheet" type="text/css"
    href="node_modules/jasmine-core/lib/jasmine-core/jasmine.css">
  <script type="text/javascript"
    src="node_modules/jquery/dist/jquery.js">
  </script>
  <script type="text/javascript"
    src="node_modules/jasmine-core/lib/jasmine-core/jasmine.js">
  </script>
  <script type="text/javascript"
    src="node_modules/jasmine-core/lib/jasmine-core/jasmine-
    html.js">
  </script>
```

```
    <script type="text/javascript"
      src="node_modules/jasmine-core/lib/jasmine-core/boot.js">
    </script>
    <script type="text/javascript"
      src="node_modules/jasmine-jquery/lib/jasmine-jquery.js">
    </script>
    <script type="text/javascript"
      src="html_spec/HtmlTests.spec.js"></script>
  </head>
  <body>
  </body>
  </html>
```

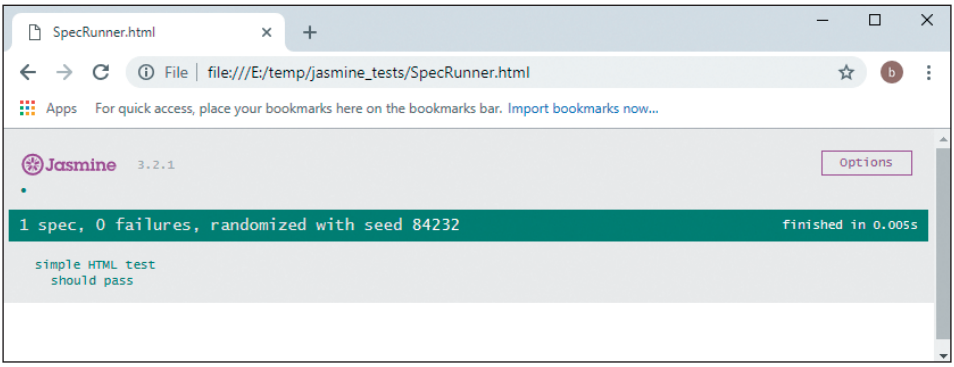
Это простая HTML-страница, которая содержит несколько исходных файлов JavaScript. При установке Jasmine предоставляет три таких файла: `jasmine.js`, `jasmine-html.js` и `boot.js`. Обратите внимание, что эти файлы должны быть включены в этом точном порядке, в противном случае страницу не удастся загрузить правильно. Другие файлы, которые включены, – это `jquery.js` и `jasminejquery.js`. Эти файлы не являются частью стандартной установки Jasmine и поэтому должны быть установлены через npm:

```
npm install jquery jasmine-jquery --save-dev
```

Последний файл, который мы включаем, – это `html_spec/HtmlTests.spec.js`, являющийся результатом компиляции нашего файла `.ts` и содержащий сами тесты. Мы можем создать этот файл и вставить простой тест, чтобы убедиться, что все работает правильно:

```
describe("simple HTML test", () => {
  it("should pass", () => {
    expect(true).toBeTruthy();
  });
});
```

После этого мы можем открыть файл `SpecRunner.html` и увидеть, что Jasmine выполняет наши тесты в браузере:



Фикстуры

Во многих случаях наш код отвечает за чтение или, в большинстве случаев, манипулирование элементами DOM из JavaScript. Это означает, что любой работающий код, который опирается на элемент DOM, может потерпеть неудачу, если базовый HTML-код не содержит правильный элемент или группу элементов. Чтобы протестировать функции, которые каким-либо образом модифицируют DOM, нам нужно будет предоставить копию или настоящие элементы DOM для прохождения тестов.

Одна из библиотек расширений Jasmine, названная `jasmine-jquery`, позволяет нам делать именно это. Библиотека `jasmine-jquery` дает возможность вставлять HTML-элементы в DOM перед выполнением наших тестов, а затем автоматически удаляет их после запуска теста. Вот почему мы включили эту библиотеку в файл `SpecRunner.html`.

Давайте рассмотрим пример класса, который модифицирует элемент DOM:

```
class ModifyDomElement {
   setHtml() {
      let elem = $('#my_div');
      elem.html('<p>Hello World</p>');
   }
}
```

У класса `ModifyDomElement` есть единственная функция `setHtml`, которая использует jQuery для поиска элемента DOM с идентификатором `my_div`. HTML-код этого `div` затем устанавливается в простой абзац "`Hello World`". Очевидно, этот класс требует наличия элемента DOM с именем `my_div` для правильной работы. Давайте теперь посмотрим, как можно использовать функцию `setFixtures` из библиотеки `jasmine-jquery` в тесте для настройки этого элемента DOM:

```
describe("fixture tests", () => {
```

```
    it("should modify a dom element", () => {
      setFixtures('<div id="my_div"></div>');
      let modifyDom = new ModifyDomElement();
      modifyDom.setHtml();
      var modifiedDomElement = $('#my_div');
      expect(modifiedDomElement.length).toBeGreaterThan(0);
      expect(modifiedDomElement.html()).toContain("Hello");
    });
  });
```

Тест начинается с вызова функции `jasmine-jquery`, `setFixtures`. Эта функция будет вставлять HTML-код, предоставленный в виде строкового параметра, непосредственно в DOM на время теста. Затем тест создает экземпляр класса `ModifyDomElement` и вызывает функцию `setHtml`, которая изменит элемент `my_div`.

В оставшейся части теста используется функция jQuery `$` для поиска элемента DOM с идентификатором `my_div`, которая сохраняется в переменной `modifiedElement`. Затем переменная `modifiedElement` передается нашим двум операторам ожидания. Обратите внимание, что первый оператор `expect` проверяет, чтобы увидеть, является ли свойство `length` переменной `modifiedDomElement` больше `0`. Это самый простой способ выяснить, действительно ли элемент был найден в DOM. Если он был найден, мы проверяем внутренний HTML-код элемента, чтобы убедиться, что он содержит строку `"Hello"`.

Методы с использованием фикстур, предоставляемые `jasmine-jquery`, также позволяют загружать сырые HTML-файлы с диска, вместо того чтобы записывать длинные строковые представления HTML-кода. Это также особенно полезно, если ваш MV*-фреймворк использует фрагменты HTML-файла. Кроме того, в `jasmine-jquery` также есть утилиты для загрузки JSON с диска и специальные средства поиска, которые работают с jQuery. Обязательно ознакомьтесь с документацией к библиотеке на странице `https://github.com/velesin/jasmine-jquery`.

События DOM

Будут моменты, когда код, который вы пишете, должен реагировать на события DOM, такие как `onclick` или `onselect`. К счастью, написание тестов, которым нужны эти события, также может быть смоделировано с использованием `jasmine-jquery` и шпионов:

```
describe("click event tests", () => {
  it("should trigger an onclick DOM event", () =>{
    setFixtures(`
      <script>
```

```
            function handle_my_click_div_clicked() {
               // do nothing at this time
            }
         </script>
         <div id='my_click_div'
         onclick='handle_my_click_div_clicked()'>Click Here</div>`);
      var clickEventSpy = spyOnEvent('#my_click_div', 'click');
      $('#my_click_div').click();
      expect(clickEventSpy).toHaveBeenTriggered();
   });
});
```

В этом тесте мы снова вызываем функцию `setFixtures` из библиотеки `jasmine-jquery`. Она делает две вещи. Во-первых, она определяет функцию `handle_my_click_div_clicked` в теге `<script>`. Во-вторых, определяет `<div>` с идентификатором `my_click_div`, а затем присоединяет DOM-событие `onclick` для вызова функции `handle_my_click_div_clicked()`. Поэтому этот единственный вызов функции устанавливает весь необходимый HTML-код для события `onclick`. Без тега `<script>` выполнение наших тестов приведет к ошибке:

```
ReferenceError: handle_my_click_div_clicked is not defined
```

Затем мы устанавливаем шпиона с именем `clickEventSpy`. Он использует функцию `spyOnEvent`, которая принимает два параметра – селектор jQuery для элемента, за которым следят, и имя события DOM.

Далее мы используем jQuery для запуска события, вызывая `$('#my_click_div').Click()`.

Помните, что поведение шпиона по умолчанию – это захватить определение функции и вместо этого вызвать нашего шпиона. Последняя строка этого теста – оператор ожидания, где мы ожидаем, что шпион был вызван. Функция `toHaveBeenTriggered` является сопоставителем Jasmine, который предоставлен библиотекой `jasmine-jquery`.

Манипуляции с JQuery и DOM дают нам способ заполнить форму, нажимая кнопки **Submit**, **Cancel** и **OK** и обычно имитируя взаимодействие пользователя с нашим приложением. С помощью этих методов мы можем легко написать полные приемочные тесты в Jasmine, что еще больше защитит наше приложение от ошибок и изменений.

Библиотеки для модульного тестирования

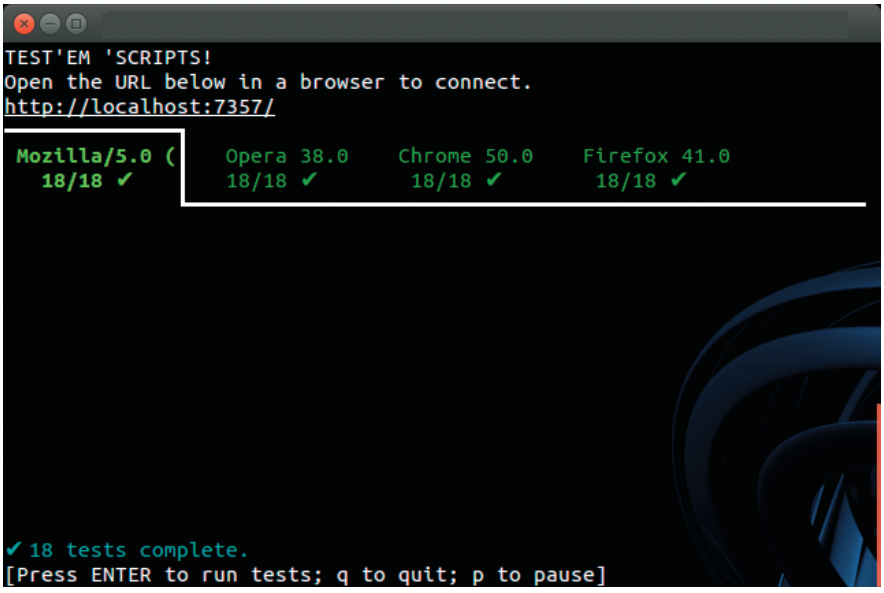
Запуск веб-страницы для запуска тестов каждый раз, когда мы вносим изменения в один из наших тестов, может быстро стать трудоемким и подверженным ошиб-

кам. Мы уже рассматривали использование Grunt в своих инструментах сборки, чтобы обнаруживать изменения файлов и автоматически перекомпилировать файлы TypeScript при сохранении файла. В этом разделе мы рассмотрим несколько библиотек для модульного тестирования, которые обнаружат изменения в нашем наборе тестов и автоматически перезапустят наши тесты без постороннего вмешательства. Использование этих библиотек дает мгновенную информацию о состоянии всех тестов, когда мы пишем свой код и сохраняем изменения.

Testem

Testem – это основанная на Node библиотека для модульного тестирования. Она запускается из командной строки и открывает простой интерфейс для просмотра результатов теста. Testem автоматически обнаружит изменения в файлах JavaScript и выполнит тесты на лету, обеспечивая мгновенную обратную связь на этапе модульного тестирования.

Testem также имеет очень удобную функцию, которая позволяет нескольким браузерам подключаться к одному экземпляру testem. Это позволяет подключить экземпляр Chrome, Firefox, IE, Opera, Safari, QupZilla или почти любого типа браузера к одной и той же библиотеке. Testem будет перезапускать наши тесты в каждом браузере и даст сводку:



Testem также имеет параметр непрерывной интеграции, который можно использовать на серверах сборки. Дополнительную информацию можно найти в репозитории GitHub (`https://github.com/airportyh/testem`).

Testem можно установить через Node с помощью следующей команды (обратите внимание, что вам может потребоваться префикс sudo в системе на основе Linux):

```
pm install -g testem
```

По умолчанию Testem попытается загрузить любые файлы JavaScript в текущем каталоге, проанализировать их для любых тестов и затем запустить их, когда подключен браузер. Поэтому Testem создает простую HTML-страницу в памяти и предоставляет эту страницу нашим браузерам. Нам нужно будет настроить Testem, создав простой файл `testem.json` в директории `test`:

```
{
   "framework": "jasmine2",
   "src_files": [
      "node_modules/jquery/dist/jquery.js",
      "node_modules/jasmine-jquery/lib/jasmine-jquery.js",
      "html_spec/HtmlTests.spec.js"
   ]
}
```

Это простой файл формата JSON, в котором указаны два свойства: `framework` и `src_files`. Свойство `framework` указывает на то, что мы используем `"jasmine2"` в качестве фреймворка, а свойство `source_files` включает в себя некоторые дополнительные файлы JavaScript, которые необходимы для наших тестов наряду с самим файлом `html_spec/HtmlTests.spec.js`. Теперь мы можем запустить наш набор тестов. Обратите внимание, что хотя мы указываем `jasmine2` в качестве нашего фреймворка, мы можем использовать Jasmine 3.0 и выше.

В Testem есть ряд мощных опций конфигурации, которые можно указать в файле конфигурации. Обязательно зайдите в репозиторий GitHub для получения дополнительной информации.

Обратите внимание, что testem – это хороший выбор для модульного тестирования, но не подходит для интеграционного или приемочного тестирования. Природа этого фреймворка означает, что Testem на лету создает HTML-страницу на основе нашего файла конфигурации. Во время интеграционного тестирования нам обычно нужны HTML-страницы, которые будут созданы веб-сервером.

Karma

Karma – это библиотека для модульного тестирования, созданная командой AngularJs. Она широко представлена в курсах по Angular. Это всего лишь фреймворк для модульного тестирования, и команда AngularJs рекомендует создавать сквозные или интеграционные тесты и запускать их через Protractor. Karma, как и Testem, запускает свой собственный экземпляр веб-сервера для обслуживания страниц и артефактов, требуемых набором тестов, и обладает большим набором

параметров конфигурации. Она также может быть использована для модульных тестов, которые не нацелены на Angular. Чтобы установить Karma для работы с Jasmine, нужно установить несколько пакетов, используя npm:

```
npm install -g karma-cli
npm install karma --save-dev
npm install karma-jasmine --save-dev
npm install karma-chrome-launcher --save-dev
npm install karma-jasmine-jquery --save-dev
npm install karma-jquery --save-dev
```

Для запуска Karma нам понадобится файл конфигурации. Мы можем создать стандартный файл `karma.conf.js`, выполнив в командной строке:

```
karma init
```

Далее последует несколько вопросов о фреймворках, которые мы используем. Продолжайте и примите значения по умолчанию на этом этапе. После того как файл по умолчанию `karma.conf.js` был создан, мы можем несколько изменить его, чтобы он запускал наши существующие тесты.

Во-первых, нам нужно изменить параметр `frameworks`:

```
frameworks: ['jquery-3.3.1', 'jasmine-jquery', 'jasmine'],
```

Здесь мы включили `jquery-3.3.1` и `jasmine-jquery` в опцию `frameworks`, чтобы мы могли использовать `jasmine-jquery` в Karma. Нам также нужно будет добавить параметр `plugins`:

```
plugins: [
   require('karma-jasmine'),
   require('karma-chrome-launcher'),
   require('karma-jasmine-jquery'),
   require('karma-jquery'),
   require('karma-spec-reporter')
],
```

Мы сообщили Karma, что ей нужно загрузить набор модулей, которые нам понадобятся в наших тестах. Модули `karma-jasmine`, `karma-jasmine-jquery` и `karma-jquery` настроят нашу среду для использования jQuery и `jasmine-jquery`. Модуль `karma-chromelauncher` используется для запуска экземпляра Chrome при запуске наших тестов. Обратите внимание, что мы также включили `karma-spec-reporter`, который является репортером Jasmine, чтобы дать нам лучшую информацию о командной строке.

Последнее изменение, которое нам нужно сделать, – это сообщить Karma, где найти наши тестовые файлы:

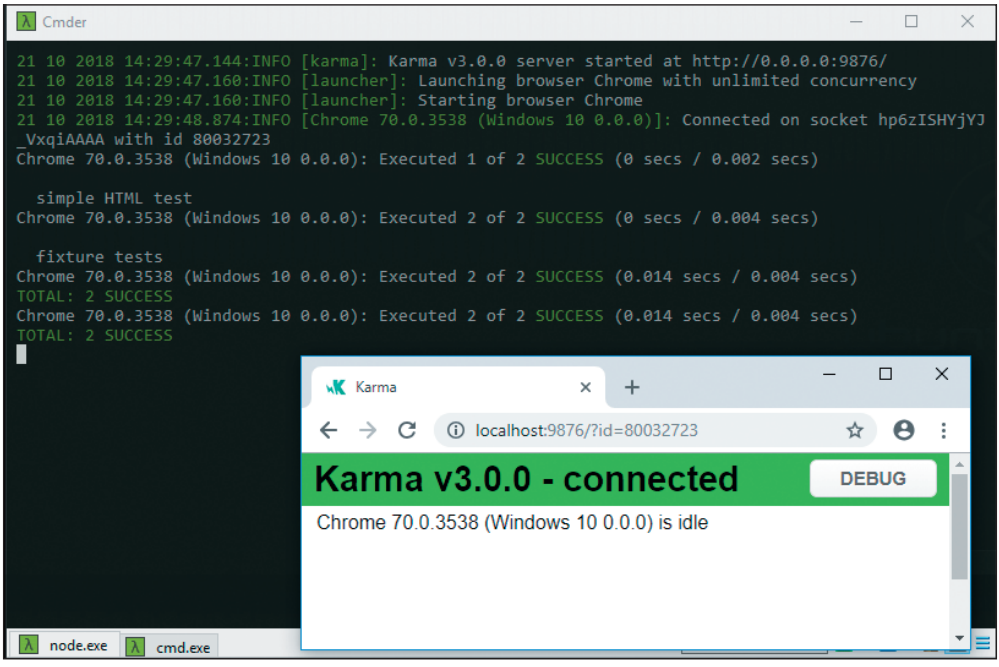
```
files: [
   'html_spec/**/*spec.js'
],
```

Здесь мы указали, что Karma должна искать в каталоге `html_spec` и включать любой файл, который оканчивается на `spec.js`.

После этих изменений в файле конфигурации мы можем запустить Karma, просто набрав:

```
karma start <path to karmea.config.js>
```

После этого произойдет запуск Karma, автоматически запустится экземпляр Chrome и все наши тесты. Обратите внимание, что Karma будет продолжать работать в фоновом режиме, и как только будут найдены изменения в любом из тестовых файлов, она перезапустит все наши тесты:



Тестирование в режиме headless

При запуске модульных тестов в среде непрерывной интеграции нам может потребоваться запускать тесты с помощью браузера, не имеющего графического интерфейса пользователя (headless bowser). В нем не открывается новое окно, и это хороший вариант для серверов сборки, на которых не установлен пользовательский интерфейс. В качестве такого браузера для проведения тестирования ис-

пользовался PhantomJs, предлагающий ES5-совместимый браузер. К сожалению, проект PhantomJs был приостановлен из-за отсутствия активного вклада. Однако в Chrome имеется режим запуска headless, что дает нам полную рабочую версию Chrome, которую мы можем запустить на серверах сборки. Чтобы запустить пакет Karma с Chrome в режиме headless, просто измените свойство `browsers` в файле `karma.config.js`:

```
browsers: ['ChromeHeadless'],
```

Эта простая настройка – все, что нам нужно для запуска Chrome в автономной конфигурации.

Protractor

Protractor – это библиотека для модульного тестирования на основе Node, которая занимается сквозным или автоматическим приемочным тестированием. В отличие от Testem и Karma, которые создают веб-страницу для модульного тестирования, Protractor используется для программного управления веб-браузером. Так же, как и при ручном тестировании, Protractor имеет возможность просматривать определенную страницу, а затем взаимодействовать с ней из JavaScript. В качестве простого примера предположим, что на вашем сайте есть страница входа в систему, а для всего остального требуется действительный вход в систему. Используя Protractor, мы можем начинать каждый тест, перейдя на страницу входа в систему, введя действительные учетные данные, а затем продолжив просмотр каждой страницы, которая является частью нашего набора тестов.

Используя Protractor, мы также можем проверять свойства метаданных на HTML-странице, такие как заголовок страницы, или можем заполнять формы и нажимать кнопки. Protractor можно установить с помощью npm:

```
npm install -g protractor
```

Мы приступим к запуску Protractor чуть позже, а сначала давайте обсудим движок, который Protractor использует под капотом для управления браузером.

Selenium

Selenium – инструмент для веб-браузеров. Он позволяет программно управлять веб-браузерами и может использоваться для создания автоматических тестов в Java, C#, Python, Ruby, PHP, Perl и даже JavaScript. Protractor использует Selenium для управления экземплярами веб-браузера. Чтобы установить Selenium Server для использования с Protractor, выполните следующую команду:

```
webdriver-manager update
```

Чтобы запустить его, выполните следующую команду:

```
webdriver-manager start
```

Если все пойдет хорошо, Selenium сообщит, что сервер запущен, и подробно опишет адрес Selenium Server. Проверьте вывод на предмет наличия там строки, похожей на эту:

```
RemoteWebDriver instances should connect to:
http://127.0.0.1:4444/wd/hub
```

Обратите внимание, что вам понадобится установить Java на вашем компьютере, так как менеджер веб-драйверов использует Java для запуска Selenium Server.

После запуска сервера нам понадобится файл конфигурации Protractor (аналогичный Karma), который по соглашению называется `protractor.conf.js`. Содержимое этого файла выглядит так:

```
exports.config = {
   seleniumAddress: 'http://localhost:4444/wd/hub',
   specs: ['protractor_tests/*.js']
}
```

Здесь мы просто присваиваем ряд свойств объекту `exports.config`. Первое свойство – это `seleniumAddress`, которое является экземпляром Selenium Server, как мы видели ранее. Второе свойство, specs, представляет собой список тестов для запуска. Это свойство ищет любые файлы `.js` в каталоге `protractor_tests`.

Далее идет самый простой из тестов:

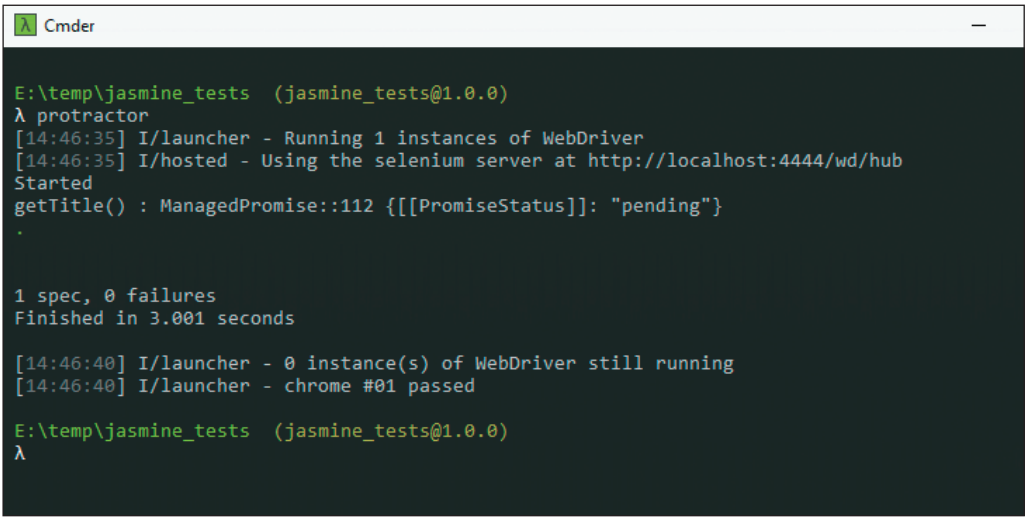
```
describe("simple protractor test", () => {
   it("should navigate to google and find a title", () => {
      browser.driver.get('http://www.google.com');
      expect(browser.driver.getTitle()).toContain("Google");
   });
});
```

Наш тест начинается с открытия страницы по адресу `http://www.google.com`. Затем мы ожидаем увидеть, что заголовок страницы установлен на «`Google`». Теперь мы можем запустить Protractor для выполнения этого теста:

```
protractor
```

Если вы будете следить за своим экраном, то увидите, что Protractor запускает новый экземпляр сеанса браузера Chrome, а после этого перейдет на главную страницу Google.

Затем он выполнит ожидание. Наш вывод командной строки выглядит так:



Поиск элементов страницы

В Selenium есть ряд функций, которые мы можем использовать для поиска HTML-элементов на странице во время тестирования. Мы можем искать элемент, используя его свойство `id`, или можем использовать CSS-селектор или `xpath`. В качестве примера рассмотрим следующий тест:

```
it('should search for the term TypeScript', async () => {
   browser.driver.get("https://www.google.com");
   await browser.driver.findElement(
      By.id("lst-ib")).sendKeys("TypeScript");
   await browser.driver.findElement(
      By.xpath('//*[@id="lst-ib"]')).sendKeys(Key.ENTER);
   browser.sleep(5000);
});
```

Мы написали тест, который переходит на страницу `www.google.com`, а затем использует функцию `findElement`. Первый вызов `findElement` использует статическую функцию `Selenium` с именем `By.id`, которая принимает строковый аргумент. Он запросит DOM и найдет элемент с соответствующим атрибутом `id`. Найдя этот элемент, мы можем смоделировать ввод данных с помощью функции `sendKeys` с эффектом ввода слова «`TypeScript`» в поле поиска на главной странице Google.

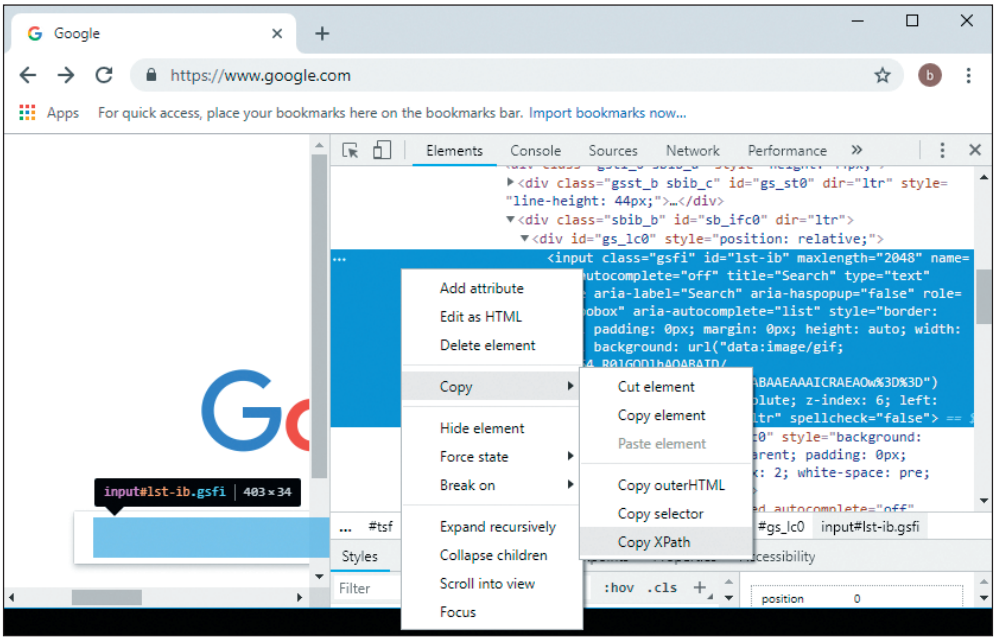
Мы также можем запросить элементы DOM, используя `xpath`. Второй вызов `findElement` в нашем тесте использует функцию `By.xpath` для выполнения запроса с использованием сопоставителя `xpath`. Обратите внимание, что в этом тесте запросы `By.id` и `By.xpath` найдут один и тот же элемент, который явля-

ется полем поиска. Мы также можем имитировать нажатие клавиш *Enter* или *Tab*, используя встроенные перечисления для специальных клавиш, в данном случае `Key.ENTER`.

Последним вызовом в нашем тесте является перерыв браузера на 5 секунд. Есть ряд функций, которые мы можем использовать в Selenium, в том числе перемещение мыши, нажатие на Elements или приостановка работы браузера в целях отладки.

Мы можем легко использовать инструменты разработчика Chrome, чтобы найти правильные элементы для использования в наших тестах Selenium либо по идентификатору, либо по css, либо по xpath. Если мы откроем инструменты разработчика с помощью клавишы **F12**, а затем щелкнем правой кнопкой мыши на интересующем нас элементе, то можем выбрать опцию меню **Inspect**, чтобы открыть инструменты разработчика и показать дерево DOM. Если затем мы щелкнем правой кнопкой мыши по конкретному элементу в дереве DOM, то можем использовать пункт меню **Copy** (**Копировать**), а затем пункт меню **Copy Selector** (**Копировать селектор**), чтобы скопировать css-селектор в буфер обмена.

Аналогичным образом мы также можем скопировать селектор xpath в буфер обмена, как показано на скриншоте ниже.



Chrome даже позволяет искать дерево DOM, используя `xpath` или css-селектор. Если мы перейдем на вкладку **Консоль**, то сможем выполнить поиск с помощью селектора `xpath`, введя следующее:

```
$x("... xpath selector goes here")
```

Или мы можем выполнить поиск по селектору CSS, набрав:

```
$("... css selector goes here")
```

Эти инструменты неоценимы, когда речь идет о поиске элементов на наших страницах при создании наборов тестов Selenium.

Использование непрерывной интеграции

При написании модульных тестов для любого приложения быстро возникает необходимость настройки сервера сборки и запуска своих тестов как части регистрации изменений в системе контроля версий. Когда ваша команда разработчиков выходит за рамки одного разработчика, использование сервера сборки непрерывной интеграции становится обязательным. Этот сервер гарантирует, что любой код, отправленный на сервер управления версиями, пройдет все известные модульные, интеграционные и автоматические приемочные тесты.

Сервер сборки также отвечает за маркировку сборки и генерацию любых артефактов, которые необходимо использовать во время развертывания. Основные шаги при настройке сервера сборки следующие:

- проверьте последнюю версию исходного кода и увеличьте номер сборки;
- скомпилируйте приложение на сервере сборки;
- запустите модульные тесты на стороне сервера;
- упакуйте приложение для развертывания;
- разверните пакет в среде сборки;
- запустите интеграционные тесты на стороне сервера;
- запустите модульные, интеграционные или приемочные тесты JavaScript;
- пометьте набор изменений и номер сборки как пройденный или непройденный;
- если сборка не удалась, сообщите об этом тем, кто ее нарушил.

Сервер сборки потерпит неудачу, если произойдет сбой любого из предыдущих шагов.

Преимущества непрерывной интеграции

Использование сервера сборки при проходе предыдущих шагов приносит огромную пользу любой команде разработчиков. Во-первых, приложение компилируется на сервере сборки – это означает, что на нем должны быть установлены

инструменты или внешние библиотеки. Это дает вашей команде разработчиков возможность задокументировать, какое именно программное обеспечение необходимо установить на новом компьютере, чтобы скомпилировать или запустить приложение.

Во-вторых, перед попыткой упаковки можно запустить стандартный набор модульных тестов на стороне сервера. В проекте Visual Studio это будут модульные тесты C#, созданные с использованием одного из популярных фреймворков тестирования .NET – MsTest, nUnit или xUnit.

После этого выполняется этап упаковки всего приложения. Давайте на минутку предположим, что разработчик включил новую JavaScript-библиотеку в проект, но забыл добавить ее в систему контроля версий. В этом случае все тесты будут выполняться на локальном компьютере, но они прервут сборку из-за отсутствующего файла библиотеки. Если на этом этапе развернуть сайт, запуск приложения приведет к ошибке **404 – file not found**. При выполнении этапа упаковки такие типы ошибок быстро обнаруживаются.

После успешного завершения этапа упаковки сервер сборки должен развернуть сайт в специально помеченной среде сборки. Эта среда сборки используется только для сборок непрерывной интеграции и поэтому должна иметь собственные экземпляры базы данных, ссылки на веб-службы и т. д., настроенные специально для сборок непрерывной интеграции. Опять же, на самом деле выполнение развертывания в целевой среде проверяет артефакты развертывания, а также процесс развертывания. Настроив среду сборки для автоматического развертывания пакетов, ваша команда снова сможет документировать требования и процесс развертывания.

На этом этапе у нас есть полный экземпляр нашего сайта, работающий в изолированной среде сборки. Затем мы можем легко нацелиться на конкретные веб-страницы, на которых будут выполняться наши тесты JavaScript, а также запускать интеграционные или автоматические приемочные тесты – непосредственно в полной версии сайта. Таким образом, мы можем написать тесты, предназначенные для реальных служб REST сайта, без необходимости макетировать эти точки интеграции. Итак, по сути, мы тестируем приложение с нуля. Очевидно, что нам может потребоваться убедиться, что в нашей среде сборки есть определенный набор данных, который можно использовать для интеграционного тестирования, или способ создания необходимых наборов данных, которые понадобятся нашим интеграционным тестам.

Выбор сервера сборки

Существует ряд серверов непрерывной интеграции, в том числе **TeamCity**, **Jenkins** и **Team Foundation Server** (**TFS**).

Team Foundation Server

TFS – это продукт Microsoft, для которого потребуется лицензия на серверный компонент, а также лицензия для разработчика. TFS нужна особая конфигурация на его агентах сборки, чтобы иметь возможность запускать экземпляры веб-браузера, так как по умолчанию она отключена. Он также использует Windows Workflow Foundation для настройки шагов сборки, для изменения которых требуется немало опыта и знаний.

Jenkins

Jenkins – это бесплатный сервер сборки непрерывной интеграции с открытым исходным кодом. У него большое сообщество и много плагинов. Установка и настройка Jenkins довольно просты. Он позволяет процессам запускать экземпляры браузера, делая его совместимым с модульными тестами JavaScript на основе браузера. Шаги сборки Jenkins основаны на командной строке, и иногда требуется правильно сконфигурировать их.

TeamCity

TeamCity – это очень популярный и очень мощный сервер сборки, который можно установить бесплатно. TeamCity допускает бесплатную установку, если у вас небольшое число разработчиков (меньше 20) и небольшое количество проектов (меньше 20). Полная коммерческая лицензия стоит всего около 1500 долларов, что делает его доступным для большинства организаций. Настроить шаги сборки в TeamCity гораздо проще, чем в Jenkins или TFS, так как он использует конфигурацию в стиле мастера в зависимости от типа создаваемого шага сборки. TeamCity также обладает богатым набором функциональных возможностей, связанных с модульными тестами, с возможностью отображать графики модульных тестов, и поэтому считается лучшим в своем классе.

Отчеты об интеграционных тестах

Мы видели, как создавать и запускать тесты с использованием Jasmine, Testem, Karma и Protractor.

Каждый из наших образцов успешно сообщил о количестве выполненных тестов, а также об успехе или неудаче набора тестов. Мы использовали простые файлы конфигурации и простые HTML-файлы для настройки и выполнения тестов.

Однако в реальных приложениях часто необходимо запускать логику на стороне сервера или использовать рендеринг HTML-кода на стороне сервера. Например, большинству приложений потребуется какая-либо аутентификация или вход в систему, прежде чем разрешать вызовы пользовательских конечных точек

служб REST через JavaScript. К сожалению, вызов любой из этих точек из обычной HTML-страницы вернет ошибку **401**. В таких случаях мы должны запускать наши тесты на полном веб-сайте.

Каждый из упомянутых нами серверов непрерывной интеграции имеет свой собственный способ сбора и создания отчетов о результатах автоматического запуска теста. Например, для сред TeamCity выходные данные тестового прогона должны соответствовать требованиям отчетов о тестировании TeamCity.

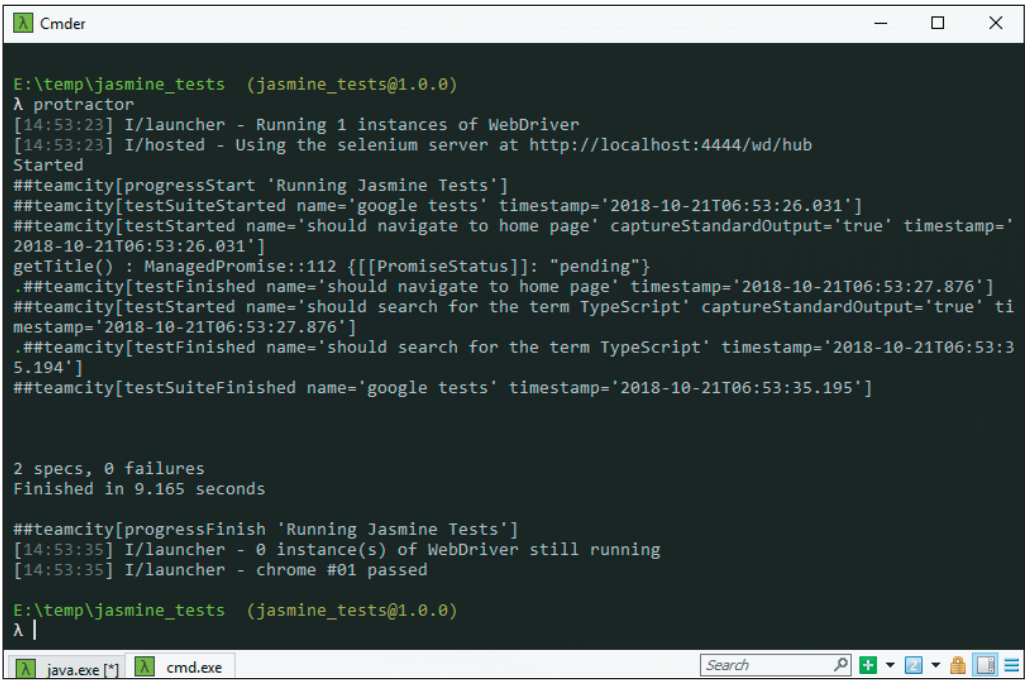
К счастью, пакет jasmine-reporters, который мы обсуждали ранее, уже поддерживает большинство серверов сборки непрерывной интеграции.

Мы можем настроить наш файл конфигурации Protractor для использования репортера Jasmine следующим образом:

```
exports.config = {
   seleniumAddress: 'http://localhost:4444/wd/hub',
   specs: ['protractor_tests/*.js'],
   onPrepare: function() {
      var jasmineReporters = require('jasmine-reporters');
      jasmine.getEnv().addReporter(
         new jasmineReporters.TeamCityReporter());
   }
}
```

Здесь мы включили свойство `onPrepare` в наши настройки конфигурации, чтобы запустить анонимную функцию. Эта функция просто создает переменную `jasmineReporters` посредством вызова `require`, а затем добавляет новый `TeamCityReporter` в среду выполнения Jasmine. Вызов функции `require` является частью механизма загрузки модуля, который мы рассмотрим в следующей главе.

При запуске тестов с Protractor теперь выводятся сообщения в командную строку, которые понимает TeamCity:



Резюме

В этой главе мы с нуля исследовали разработку через тестирование. Мы обсудили теорию этого вида разработки, изучили различия между модульными, интеграционными и приемочными тестами и взглянули на то, как выглядит процесс сервера непрерывной интеграции. Затем мы изучили Jasmine в качестве фреймворка тестирования, научились писать тесты, использовали ожидания и сопоставления, а также исследовали расширения Jasmine, чтобы облегчить работу с тестами на основе данных и работу с тестами DOM с помощью фикстур. Наконец, мы взглянули на библиотеки для модульного тестирования, обсудили, где и когда их лучше всего использовать, и использовали Protractor для управления веб-страницами через Selenium и передачи результатов обратно на сервер сборки.

В следующей главе мы рассмотрим, как создавать тесты для фреймворков, совместимых с TypeScript: Backbone, Aurelia, Angular и React.

Глава 9

Тестирование фреймворков, совместимых с Typescript

В главе 7 *«Фреймворки, совместимые с TypeScript»* мы обсудили совместимые с TypeScript фреймворки и выяснили, как Backbone, Aurelia, Angular и React используют шаблоны проектирования MVC или MV* для написания моделей, представлений и контроллеров. Мы реализовали одно и то же приложение в каждом из этих фреймворков, чтобы иметь возможность сравнить сходства между ними и отметить тонкие различия. Затем в нашей последней главе мы начали исследовать разработку через тестирование и обсудили использование Jasmine в качестве фреймворка для тестирования. Мы также изучили возможность использования различных библиотек для модульного тестирования, включая Testem и Karma, и, наконец, изучили Protractor для выполнения интеграционных или сквозных тестов.

В этой главе мы, по сути, объединим нашу работу из предыдущих двух глав и будем обсуждать, как выполнить модульное и интеграционное тестирование с использованием каждого из наших фреймворков, совместимых с TypeScript. Затем для каждого из этих фреймворков мы рассмотрим следующие темы:

- настройка фреймворка для модульного тестирования;
- написание базовых модульных тестов;
- проверка DOM для визуализированных элементов;
- проверка начальных значений формы;
- изменение значений формы;
- отправка форм.

Тестирование нашего приложения

Вы помните, что у нашего приложения были следующие особенности:

- использование представления для отображения свойства модели;
- создание массива данных, где каждый элемент массива является единичным экземпляром модели;
- перебор элементов массива и визуализация элемента <button> для каждого элемента;

- ответ на событие click для каждого элемента кнопки;
- показ выбранного элемента;
- отображение формы с полем ввода, с предварительно установленным значением;
- ответ на событие отправки формы.

Если бы нам пришлось вкратце обрисовать тесты, которые мы могли бы написать, в идеале нам бы хотелось, чтобы они охватывали следующие сценарии:

- **тесты моделей**: эти тесты охватывают создание и использование моделей в рамках нашего приложения;
- **тесты состояния приложения**: эти тесты охватывают значения по умолчанию или состояние создаваемых элементов;
- **тесты визуализации**: эти тесты будут запрашивать элементы DOM, которые визуализируются для каждого элемента приложения, и обеспечивать их правильное подключение к DOM;
- **тесты событий DOM**: эти тесты гарантируют, что приложение правильно реагирует на события DOM, такие как нажатие элемента;
- **тесты форм**: эти тесты гарантируют, что элементы формы создаются с правильными значениями по умолчанию и что значения форм читаются правильно после ввода пользователя;
- **тесты отправки формы**: эти тесты гарантируют, что при отправке формы вызываются правильные функции.

Тестирование Backbone

В этом разделе мы рассмотрим тесты, которые нам нужно написать, чтобы охватить необходимые функциональные возможности нашего приложения для Backbone.

Настройка теста

Одним из преимуществ написания приложений в Backbone является то, что сам фреймворк не имеет много зависимостей. Чтобы запустить приложение Backbone, нам нужно загрузить библиотеку Underscore, библиотеку jQuery (или эквивалент) и сам фреймворк Backbone.

Как только эти библиотеки будут загружены в наш браузер, нам просто нужно включить все файлы, которые мы написали для приложения. В качестве примера рассмотрим тег `<head>` в нашем файле `index.html`:

```
<head>
  <link rel="stylesheet" type="text/css"
    href="./node_modules/bootstrap/dist/css/bootstrap.css">
```

```
    <script src="./node_modules/underscore/underscore.js"></script>
    <script src="./node_modules/jbone/dist/jbone.js"></script>
    <script src="./node_modules/backbone/backbone.js"></script>
    <script src="./models.js"></script>
    <script src="./views.js"></script>
    <script src="./app.js"></script>
  </head>
```

Здесь мы включили файлы `bootstrap.css`, `underscore.js`, `jbone.js` и `backbone.js`, как было упомянуто ранее. Затем мы включили файлы `models.js`, `views.js` и `app.js`, чтобы браузер загрузил все исходные файлы, необходимые для запуска приложения. Поскольку файлов так мало, мы можем легко настроить среду тестирования с помощью Testem, создав файл `testem.json` в корневом каталоге проекта:

```
{
   "framework": "jasmine2",
   "src_files": [
      "node_modules/jquery/dist/jquery.js",
      "node_modules/jasmine-jquery/lib/jasmine-jquery.js",
      "node_modules/underscore/underscore.js",
      "node_modules/backbone/backbone.js",
      "models.js",
      "views.js",
      "app.js",
      "models.spec.js"
   ]
}
```

Здесь у нас есть простой конфигурационный файл Testem, определяющий каждый из файлов, которые нужны приложению, используя свойство `src_files`. Файлы, перечисленные в этой конфигурации, соответствуют файлам, которые мы видели в `index.html` ранее, с добавлением файла `jasminejquery.js`, который будет использоваться для настройки кода фикстуры. Единственный другой файл, который мы перечислили здесь, – это сам тестовый файл с именем `models.spec.js`, в котором будет содержаться наш тестовый код. Мы можем запустить наши тесты, запустив `testem` в командной строке, а затем подключив экземпляр браузера к `localhost: 7357`, как мы это делали в предыдущей главе.

Тесты моделей

Модели Backbone представляют собой основной метод хранения состояний в рамках нашего приложения. Когда мы создаем представление, то основываем его на данных, содержащихся в базовых моделях. Когда мы реагируем на взаимодействие с пользователем, то сохраняем результаты этого взаимодействия в модели, поэтому правильная установка и получение значений из наших моделей явля-

ются фундаментальной особенностью нашего приложения. В производственных приложениях эти модели обычно создаются или обрабатываются из данных в формате JSON, которые извлекаются из конечной точки REST. Поэтому важно проверить, что наши модели могут быть созданы правильно и получение или установка значений в нашей модели работает, как задумано.

Когда мы конструируем модель Backbone, то используем POJO внутри конструктора, чтобы присвоить значения каждому из свойств модели. Следовательно, когда мы сталкиваемся с таким интерфейсом:

```
interface IClickableItem {
   DisplayName: string;
   Id: number;
}
```

Мы создаем новый экземпляр нашего класса `ItemModel` следующим образом:

```
itemModel = new ItemModel({Id : 1, DisplayName: 'testDisplay'});
```

Помните, что Backbone хранит эти значения POJO как атрибуты в самом экземпляре класса, что приводит нас к некоторому шаблонному коду при написании версии `Backbone.Model` для TypeScript, как показано ниже:

```
class ItemModel extends Backbone.Model implements IClickableItem {
   get DisplayName(): string
      { return this.get('DisplayName'); }
   set DisplayName(value: string)
      { this.set('DisplayName', value); }
   get Id(): number { return this.get('Id'); }
      set Id(value: number) { this.set('Id', value); }
   constructor(input: IClickableItem) {
      super();
      this.set(input);
   }
}
```

Каждое свойство в нашем интерфейсе (в данном случае `IClickableItem`) должно определять пару функций `get` и `set` и использовать функции `this.get` или `this.set` для правильного хранения этих свойств. Поскольку мы пишем код, чтобы это было сделано, нам нужно написать модульные тесты, дабы убедиться, что все работает правильно.

Наш начальный набор модульных тестов выглядит так:

```
describe('ItemModel tests', () => {
   let itemModel: ItemModel;
   beforeEach( () => {
      itemModel = new ItemModel(
```

```
        {Id: 10, DisplayName: 'testDisplayName'}
      );
    });
    it('should assign an Id property', () => {
      expect(itemModel.Id).toBe(10);
    });
    it('should assign a DisplayName property', () => {
      expect(itemModel.DisplayName).toBe('testDisplayName');
    });
});
```

Здесь мы определяем переменную для хранения экземпляра `ItemModel` с именем `itemModel`.

Обратите внимание, что его определение находится за пределами функции `beforeAll`, и поэтому оно доступно для каждого из наших модульных тестов. Наша функция `beforeEach` инициализирует экземпляр класса `ItemModel` со значениями по умолчанию для повторного использования каждого из наших тестов.

Первый тест под названием '`should assign an Id property`' проверяет, что свойство `Id` возвращает то же значение, которое использовалось в конструкторе, которое в данном случае равно `10`.

Аналогично, у нас есть еще один тест для свойства `DisplayName`, и мы можем проверить, что его значение на самом деле равно '`testDisplayName`'.

Теперь мы можем расширить эти тесты, чтобы убедиться, что функции set работают правильно:

```
it('should set an Id property', () => {
  itemModel.Id = -10;
  expect(itemModel.Id).toBe(-10);
});
it('should set a DisplayName property', () => {
  itemModel.DisplayName = 'updatedDisplayName';
  expect(itemModel.DisplayName).toBe('updatedDisplayName');
});
```

В качестве дополнительного набора тестов мы можем даже обойти функции `set` и `get` и убедиться, что базовые функции Backbone также устанавливают свойства правильно:

```
it('should update the Id property when calling calling set', () => {
  itemModel.set('Id', 99);
  expect(itemModel.Id).toBe(99);
});
```

```
it('should update the DisplayName property when calling set', () => {
   itemModel.set('DisplayName', 'setDisplayName');
   expect(itemModel.DisplayName).toBe('setDisplayName');
});
```

Здесь мы проверяем, что внутренние функции `set` и `get` делают то же самое, что и синтаксис методов получения и установки в TypeScript.

Тесты сложных моделей

Мы можем использовать те же методы для проверки правильности создания сложных моделей.

Рассмотрим следующий набор тестов:

```
describe("model.spec.ts : ItemCollectionViewModel tests", () => {
   let itemCollectionViewModel: ItemCollectionViewModel;
   beforeEach(() => {
      itemCollectionViewModel = new ItemCollectionViewModel({
         Title: "testTitle",
         SelectedItem: {
            Id: 10,
            DisplayName: "testDisplayName"
         },
         Name: "testName"
      });
   });
```

Здесь мы создаем экземпляр нашей сложной модели с именем `itemCollectionViewModel`. Интересным моментом является создание этой сложной модели с использованием простого Java-объекта POJO. Мы вызываем конструктор в функции `beforeEach` и просто вкладываем эти объекты друг в друга. Мы устанавливаем свойство `Title`, а затем свойство `SelectedItem` для другого объекта POJO, у которого есть свойства `Id` и `DisplayName`. В конце мы устанавливаем свойство Name.

Эти объекты используют ту же структуру, что и данные, которые, как мы ожидаем, будут возвращены в формате JSON из конечных точек служб REST программно-аппаратной части сервиса. Определение таких объектных тестов может легко распространяться на интегрированные тесты, которые будут вызывать реальную веб-службу и выполнять повторную гидратацию наших моделей из простых Java-объектов.

Наши модульные тесты этой сложной модели могут затем просто просмотреть доступные свойства, чтобы убедиться, что все установлено правильно:

```
it("should set the Title property", () => {
   expect(itemCollectionViewModel.Title).toBe("testTitle");
});
it("should set the SelectedItem.Id property", () => {
   expect(itemCollectionViewModel.SelectedItem.Id).toBe(10);
});
it("should set the SelectedItem.DisplayName property", () => {
   expect(itemCollectionViewModel.SelectedItem.DisplayName)
      .toBe("testDisplayName");
});
it("should set the Name property", () => {
   expect(itemCollectionViewModel.Name).toBe("testName");
});
```

Наш первый тест проверяет значение свойства `Title`, а затем следующие тесты проверяют значение свойства `SelectedItem` (которое является дочерней моделью Backbone) и `Name`.

Использование таких простых тестов также может помочь в тестировании сценариев, в которых некоторые свойства не установлены. Другими словами, когда сложные модели гидратируются из конечной точки REST, случается, что некоторые свойства просто опускаются. Этот сценарий довольно распространен в конечных точках REST, где возвращаемая структура JSON может меняться в зависимости от варианта использования. Поэтому наши тесты могут удовлетворить эти изменения и гарантировать, что все функциональные возможности работают должным образом для различных комбинаций свойств.

Тесты визуализации

Если мы довольны тем, что наши модели Backbone работают правильно, то можем обратить внимание на их представления и написать несколько тестов, чтобы убедиться, что они правильно визуализируют эти свойства модели в DOM. Мы будем использовать функцию Jasmine, `setFixtures`, чтобы настроить шаблоны Backbone:

```
describe("views.spec.ts : ItemView tests", () => {
   let itemModel: ItemModel;
   beforeEach(() => {
      jasmine.getFixtures().fixturesPath = "./";
      loadFixtures("views.spec.html");
      itemModel = new ItemModel({
         Id: 10,
         DisplayName: "testDisplayName"
      });
   });
```

Здесь мы создали экземпляр `ItemModel` с именем `itemModel` для повторного использования в каждом тесте. Мы также вызываем функцию Jasmine, `loadFixtures`, чтобы вставить HTML-теги `<script>`, которые нам потребуются в DOM. Обратите внимание, что мы используем функцию `loadFixtures` вместо функции `setFixtures`, которую мы использовали ранее, что позволяет нам загружать HTML-код фикстуры из файла на диск. Однако, чтобы использовать эту функцию, нам нужно сообщить Jasmine, где находится файл, и правильно установить свойство `fixturesPath`, используя функцию `jasmine.getFixtures()`. Файл `views.spec.html` содержит те же HTML-шаблоны, которые мы использовали в нашем файле `index.html`.

Тег `<script>`, который будет использовать `ItemView`, – это сценарий `itemViewTemplate`, который определяет следующий HTM-код:

```
<button style="margin: 5px;" class="btn btn-primary" >
   <%= DisplayName %>
</button>
```

Этот шаблон использует свойство `DisplayName` экземпляра `ItemModel` внутри тега `<button>`.

Поэтому наш тест будет искать эти HTML-элементы после визуализации `ItemView`:

```
it("should render an ItemView correctly", () => {
   let itemView = new ItemView({ model: itemModel });
   let renderedHtml = itemView.render().el;
   expect(renderedHtml.innerHTML).toContain(
      `<button style="margin: 5px;" class="btn btn-primary">`);
   expect(renderedHtml.innerHTML).toContain(
      `testDisplayName`);
   expect(renderedHtml.innerHTML).toContain(
      `</button>`);
});
```

В этом тесте мы создаем экземпляр класса `ItemView` и создаем его, используя нашу модель. Затем мы вызываем функцию `render` для экземпляра `itemView` и сохраняем значение свойства `el` в переменную `renderedHtml`. Свойство `el` – это то, что будет присоединено к DOM и содержит HTML-код, сгенерированный в результате функции `render`.

Затем наш тест проверяет наличие элемента `<button>` и правильность замены `<% = DisplayName%>`.

Таким образом, наши тесты сделали следующее:

- создали экземпляр ItemModel;
- создали экземпляр ItemView, используя ItemModel;

- вызвали функцию `render` для ItemView;
- проверили, что визуализированный HTML-код содержит правильные элементы.

Тесты событий DOM

Следующим набором функций в нашем приложении, которые нам нужно будет протестировать, являются события DOM. Основная последовательность этих испытаний заключается в следующем:

- создать экземпляр `ItemView` с соответствующим ему `ItemModel`;
- визуализировать HTML-код;
- найти элемент <button> и смоделировать событие `click`;
- убедиться, что вызвана функция `onClicked ItemView`;
- убедиться, что функция `onClicked` вызывает сообщение шины событий.

Наш первый тест будет имитировать событие `click` и использовать шпиона Jasmine для функции `onClicked ItemView`:

```
it("should trigger onClicked event", () => {
   let clickSpy = spyOn(ItemView.prototype, 'onClicked');
   let itemView = new ItemView({ model: itemModel });
   itemView.render();
   let itemButton = itemView.$el.find(`button`).trigger('click');
   expect(clickSpy).toHaveBeenCalled();
});
```

В первой строке этого теста используется функция `spyOn` из Jasmine, чтобы прикрепить шпиона к функции `onClicked` нашего `ItemView`. Однако обратите внимание, что мы указываем `ItemView.prototype` в качестве входных данных для функции `spyOn`. Помните, что когда создается представление Backbone, мы указали через свойство `options.events`, какие функции следует привязывать к событиям DOM. К тому времени, когда мы завершили работу конструктора, мы не можем затем прикрепить шпиона к этой функции (так как он уже связан с событием DOM). Поэтому решение состоит в том, чтобы связать представление `prototype` до того, как будет создано фактическое представление.

Теперь, когда у нас есть шпион, мы можем создать представление и вызвать функцию `render`. После вызова этой функции мы можем использовать стандартный поиск JQuery DOM в представлении и инициировать щелчок, как показано в следующей строке:

```
let itemButton = itemView.$el.find(`button`).trigger('click');
```

Здесь мы используем свободный синтаксис и функцию jQuery $, чтобы найти элемент кнопки и вызвать событие `click`.

Наш тест проходит, так как событие `click` вызывает функцию `onClicked ItemView`.

Последний тест, который нам нужно построить, – это проверка правильности работы шины сообщений. Помните, что при щелчке элемента `ItemView` он вызывает событие в шине сообщений и включает его свойства модели как часть этого сообщения. На другой стороне шины сообщений `ItemCollectionView` прослушивает это событие и обновляет DOM, чтобы показать наш текущий выбранный элемент.

Поэтому наш тест выглядит так:

```
it("should trigger an event bus event", () => {
   let eventTriggered = false;
   EventBus.Bus.on("item_clicked", () => {
      eventTriggered = true;
   });
   let itemView = new ItemView({ model: itemModel });
   itemView.render();
   let itemButton = itemView.$el.find(`button`).trigger('click');
   expect(eventTriggered).toBeTruthy();
});
```

В этом тесте мы начинаем с установки флага `eventTriggered`, который имеет значение `false`.

Затем мы вызываем функцию `on` объекта `Event.Bus`, чтобы зарегистрировать прослушиватель шины событий. Он будет прослушивать сообщения типа `"item_clicked"` и при получении обновит флаг `eventTriggered`, чтобы указать, что это сообщение было получено. Затем мы создаем `ItemView`, как мы это делали ранее, и запускаем нажатие кнопки через DOM. Наш тест ожидает, что флаг `eventTriggered` примет значение `true`. После этого теста мы знаем, что `ItemView` правильно визуализирует элементы в DOM и что эти элементы реагируют на события щелчка, как было запланировано, а сообщения помещаются в шину событий.

Тесты представления коллекции

Наше тестирование до сих пор было сосредоточено на моделях, которые мы будем использовать в нашем приложении, и на представлении `ItemView`, которое отвечает за визуализацию одной кнопки. Теперь нам нужно расширить наши тесты до класса `ItemCollectionView`, чтобы гарантировать, что этот класс визуализирует элементы в DOM правильно и что он может правильно работать с нашей формой. Давайте начнем с настройки нашего набора тестов:

```
describe("views.spec.ts : ItemCollectionView tests", () => {
   let renderedHtml: JQuery<HTMLElement>;
```

```
    let submitSpy: jasmine.Spy;
    let itemCollectionView: ItemCollectionView;
    beforeEach(() => {
      jasmine.getFixtures().fixturesPath = "./";
      loadFixtures("views.spec.html");
      let itemCollection = new ItemCollection(ClickableItems);
      let collectionViewModel = new ItemCollectionViewModel({
        Title: "testItemCollection Title",
        SelectedItem: {
          Id: 10, DisplayName: "testSelectedItemDisplayName"
        },
        Name: "testName"
      });
      submitSpy = spyOn(ItemCollectionView.prototype
        , 'submitClick').and.callThrough();
      itemCollectionView = new ItemCollectionView({
        model: collectionViewModel
      }, itemCollection);
      renderedHtml = itemCollectionView.render().$el;
});
```

Здесь довольно много кода, но нет ничего такого, чего мы не видели раньше. Мы начинаем тестовый набор с объявления трех переменных для хранения информации, которая понадобится каждому тесту. Это `renderedHtml`, которая будет содержать HTML-код, который визуализируется `ItemCollectionView`. Если мы ищем элементы DOM, то будем использовать эту переменную в качестве отправной точки. Переменная `submitSpy` используется для хранения нашего шпиона Jasmine для функции `submitClick`, которая вызывается при отправке формы. Обратите внимание, что он использует функцию `.and.callThrough`, чтобы разрешить выполнение базового кода.

Переменная `itemCollectionView` содержит экземпляр самого `ItemCollectionView`, поэтому мы можем запрашивать его свойства.

Функция `beforeEach` устанавливает предварительные условия для теста. Во-первых, она загружает необходимые элементы DOM через вызов `loadFixtures`, а затем создает экземпляр класса ItemCollection и экземпляр класса `ItemCollectionViewModel`.

После этого тестовая установка создает нашего шпиона Jasmine для функции `submitClick`, создает экземпляр класса `ItemCollectionView` и, наконец, вызывает функцию `render` для `itemCollectionView` для хранения визуализированного HTML-кода. После установки мы можем протестировать различные элементы DOM:

```
  it("should render <h1> tag correctly", () => {
    let h1Tag = renderedHtml.find('h1');
```

```
    expect(h1Tag.html()).toContain('testItemCollection Title');
});
```

Здесь у нас тест, который проверяет, правильно ли визуализируется свойство `Title` элемента `ItemCollectionView` в теге `<h1>`. Наш следующий тест выглядит так:

```
it("should render id=ulRegions correctly", () => {
    let ulRegions = renderedHtml.find('#ulRegions');
    let buttons = ulRegions.find('button');
    expect(buttons.length).toBe(3, 'should find 3 buttons');
    let firstButton = buttons[0].innerHTML;
    expect(firstButton).toContain('firstItem');
});
```

Этот тест проверяет, что три кнопки были визуализированы в элементе `<div>` с идентификатором `ulRegions`. Затем мы также выбираем первую кнопку в этом массиве, чтобы убедиться, что свойство `DisplayName` визуализировано правильно.

Мы можем продолжить писать тесты, которые проверяют, что элемент `selectedItem` содержит свойство `DisplayName` по умолчанию и что группа форм правильно визуализирована в DOM.

Мы не будем обсуждать здесь эти тесты, но обязательно посмотрите код примеров этих тестов. Окончательный набор тестов, который мы обсудим, – это тесты для нашей формы.

Тесты формы

При работе с формами в наших приложениях обычно нужно проверить две вещи. Первое – это набор значений в форме, а второе – получение этих значений, после того как пользователь отправил форму. Помните, что в больших приложениях пользователь обычно либо вводит новые данные в форму (для создания чего-либо), либо может изменять существующие данные через форму (для обновления чего-либо). Вот почему нам нужно проверить установку начальных значений формы. Наш тест выглядит так:

```
it("should find form-group input with value", () => {
    let formGroup = renderedHtml.find('.form-group');
    let input = formGroup.find('input');
    expect(input.length).toBeGreaterThan(0,
        'could not find form-group label');
    expect(input.attr('value')).toContain('testName');
});
```

Мы находим элемент form-group в DOM с помощью CSS-селектора и сохраняем HTML-код в переменной `formGroup`. Затем мы находим элемент `input` в этой

форме и проверяем атрибут `value`, чтобы убедиться, что он был создан с правильным значением по умолчанию. Как только мы убедимся, что ввод формы заполнен правильно, мы можем написать тест, который вводит что-то в поле ввода и отправляет форму:

```
it("should enter text into input and submit the form", () => {
   let formGroup = renderedHtml.find('.form-group');
   let input = formGroup.find('input');
   expect(input.length).toBeGreaterThan(0,
      'could not find form-group label');
   // Моделируем ввод в элемент управления;
   input.val('test');
   // Находим и кликаем кнопку Submit;
   let submit = renderedHtml.find('#submit-button-button');
   expect(submit.length).toBe(1, 'could not find submit button');
   submit.click();
   // Проверяем, что функция submitClicked была вызвана;
   expect(submitSpy).toHaveBeenCalled();
   // Проверяем, что свойство было обновлено;
   expect(itemCollectionView.inputNameValue).toBe('test');
});
```

Тест начинается с поиска элемента `form-group` с использованием CSS-селектора, как мы делали раньше. Затем мы находим элемент `input` в этом HTML-коде и вызываем функцию `val`, чтобы установить значение элемента `input`. Этот вызов, то есть `input.val('test')`, – то, что имитирует пользовательский ввод в элемент формы. После того как мы успешно установили значение элемента `input`, мы ищем кнопку **Submit** по идентификатору и нажимаем на нее. Нажатие на эту кнопку вызовет функцию `submitClicked` класса `ItemCollectionView`, которая затем сможет опросить каждое поле формы и извлечь значения, введенные пользователем. Обратите внимание, что мы затем проверяем значение свойства `inputNameValue` нашего представления, чтобы удостовериться, что оно было обновлено правильно. Нам понадобится небольшая модификация класса `ItemCollectionView`, чтобы установить это свойство:

```
inputNameValue: string | undefined;

submitClick() {
   let name = this.$el.find('#inputName');
   if (name.length > 0) {
      this.inputNameValue = <string | undefined>name.val();
   } else {
      this.inputNameValue = undefined;
   }
}
```

Здесь мы добавили внутреннее свойство с именем `inputNameValue` в класс `ItemCollectionView`, в котором будет храниться значение, введенное нашим пользователем. Обратите внимание, что мы определили это поле как строковое или неопределенное. Функция `submitClick` также немного обновлена, чтобы сохранить значение элемента ввода в этой переменной.

Наш тестовый набор Backbone завершен.

Резюмируя

Как мы уже видели, при тестировании приложений Backbone мы можем выполнять широкий спектр модульных тестов, используя только Jasmine и Jasmine-jQuery. Мы можем создавать тесты моделей, тесты визуализации представлений и даже тесты событий DOM, не выходя из среды Jasmine. Мы создали и протестировали функциональность каждого из наших классов, чтобы убедиться, что они отображают правильные элементы в DOM и что мы можем взаимодействовать с формой, моделируя ввод данных пользователем, отправляя форму, а затем запрашивая значения формы для использования в рамках нашего представления.

Тестирование Aurelia

В этом разделе мы рассмотрим возможности модульного тестирования фреймворка Aurelia. Мы напишем несколько модульных тестов для созданного нами приложения, чтобы гарантировать, что начальное состояние приложения правильное, а также чтобы убедиться, что визуализированный HTML-код верен. Эти тесты будут следовать общим сценариям, которые мы изложили в начале главы.

Настройка

Один из вопросов, которые задает интерфейс командной строки Aurelia при настройке нового приложения Aurelia (`au new`), заключается в том, следует ли настраивать модульное тестирование. Если мы отвечаем «да» на этот вопрос, то все тестовые конфигурационные файлы и зависимости устанавливаются автоматически. В интересах времени мы не будем исследовать, как ретроспективно добавлять возможности модульного тестирования в существующее приложение Aurelia, а вместо этого предположим, что это уже настроено.

Чтобы запустить модульные тесты Aurelia, просто введите следующее:

```
au karma
```

После этого будет вызвана встроенная библиотека для модульного тестирования Karma и будут выполнены любые тесты, найденные в каталоге `/test/unit`, которые соответствуют соглашению имен файлов `*spec.js`.

Команда `au build` должна быть выполнена до того, как какие-либо тесты будут скомпилированы и включены в новый тестовый прогон. Aurelia предоставляет аргумент командной строки `--watch`, который автоматически перевыполнит текущую команду при обнаружении изменений в файлах на диске. Это означает, что после запуска `au karma --watch` будут автоматически скомпилированы и перезапущены любые модульные тесты Karma при изменении наших исходных файлов TypeScript. Это очень полезная функция, которая обеспечивает мгновенную обратную связь при написании модульных тестов.

Модульные тесты

Наш первый набор модульных тестов должен будет проверить, что класс App (наша точка входа) был создан правильно. Когда приложение загружается впервые, оно визуализирует заголовок, три кнопки, а также указывает, что ни один элемент не был выбран. Эти HTML-элементы связаны в свойствах `Title`, `SelectedItem` и `items` самого класса App. Давайте создадим файл `app.spec.ts` в каталоге `/test/unit` и напишем тест, чтобы убедиться, что эти свойства были установлены правильно:

```
describe('/test/unit/app.spec.ts : App tests', () => {
   let app: App;
   beforeAll(() => {
      app = new App();
   });
   it('should set Title property ', () => {
      expect(app.Title).toBe('Please select :');
   });
});
```

В первой строке этого теста используется оператор импорта для импорта класса App из файла `../../src/app`. Обратите внимание, что ссылка `../../` необходима, потому что любой оператор импорта, использующий путь, должен указывать путь, соответствующий файлу, который выполняет импорт.

Затем мы используем стандартный синтаксис `describe` от Jasmine для настройки набора тестов и настраиваем переменную `app` для хранения экземпляра класса App. Наш первый тест проверяет, что свойство `Title` класса App (в разработке) содержит фразу "`Please select`".

Поэтому этот тест проверяет начальное состояние свойства `Title` класса App при его создании. После этого мы можем проверить каждую из остальных внутренних переменных:

```
it('has a property named items', function () {
   expect(application.items).toBeDefined(); });
```

```
it('has an array of clickable items', function () {
   expect(application.items.length).toBe(3);
});
it('sets currentElement property in constructor', function () {
   expect(application.currentElement).toBeDefined();
});
it('sets currentElement.idValue to 0', function () {
   expect(application.currentElement.idValue).toBe(0);
});
it('sets currentElement.displayName to none', function () {
   expect(application.currentElement.displayName).toBe('none');
});
```

Здесь у нас есть несколько тестов для переменной `items`, которая представляет собой массив длиной 3, и несколько тестов для переменной `SelectedItem`, для которой должно быть установлено значение `Id = 0` и `DisplayName` = 'None selected'. Эти тесты проверяют, что при создании экземпляра приложения начальное состояние класса устанавливается правильно.

Тесты визуализации

Наш следующий раунд тестирования будет охватывать элементы визуализации в DOM. Aurelia предоставляет набор вспомогательных классов, аналогичных функциональности `setFixture` в Jamine, для присоединения HTML-кода к DOM и визуализации представлений с использованием этих временных элементов DOM. Поэтому наш набор тестов должен включать в себя два оператора импорта в верхней части файла:

```
import {StageComponent} from 'aurelia-testing';
import {bootstrap} from 'aurelia-bootstrapper';
```

Эти операторы импорта включают в себя класс `StageComponent` и функцию `bootstrap`. Класс `StageComponent` – это вспомогательная утилита, которую Aurelia предоставляет для размещения экземпляра нашего тестируемого элемента, который в нашем случае является экземпляром класса `App`. Функция `bootstrap` – это стандартный метод создания и начальной загрузки приложения Aurelia.

Для начала у нас есть это:

```
var app: any;

beforeEach(() => {
   app = StageComponent.withResources('app')
      .inView(`<h1>` + '${Title}' + `</h1>`)
      .boundTo(new App());
});
```

```
afterEach(() => {
   app.dispose();
});
```

Здесь мы определили переменную app для размещения экземпляра класса StageComponent. Несмотря на то что эта переменная будет содержать экземпляр класса App, мы можем установить для нее только тип any. К сожалению, попытка использовать правильный тип для переменной app приведет к ошибкам компиляции. При создании класса StageComponent используется свободный стиль, чтобы эффективно связать воедино три команды – withResources, inView и boundTo. Вызов функции withResources регистрирует app, используя StageComponent, а функция inView определяет HTML DOM, который нам нужен для наших тестов. Функция inView очень похожа на функцию setFixtures, которую мы использовали в Jasmine, но есть одно существенное отличие. Как правило, когда нам нужно включить большие фрагменты HTML-кода, мы можем просто обернуть все это в два обратных знака, то есть ` и `. К сожалению, Aurelia использует синтаксис ${<propertyName>} в HTML-шаблонах для обозначения подстановки параметров. Это противоречит стандартному синтаксису строковых литералов в TypeScript. Поэтому всякий раз, когда нам нужен тег ${<propertyName>}, нам нужно будет закрыть обратную черту и вручную добавить тег Aurelia в одинарные кавычки:

```
`<h1>` + '${Title}' + `</h1>`
```

Последний вызов функции boundTo создает новый экземпляр класса App и привязывает его к StageComponent.

Обратите внимание, что мы также определили функцию afterEach, которая вызывает метод dispose для переменной app. Это необходимо для того, чтобы элементы DOM, созданные с помощью StageComponent, корректно очищались после каждого теста.

Наш первый тест должен проверить, что свойство Title класса App правильно визуализировано в DOM:

```
it('should render Title property', (done) => {
   app.create(bootstrap).then(() => {
      const titleElement = document.querySelector("h1");
      expect(titleElement).toBeDefined();
      expect(titleElement.innerHTML).toContain('Please select:');
      done();
   }).catch(e => {
      // обратите внимание, что это дает более
      // качественные сообщения об ошибках
      console.log(`error : ${e}`);
   });
});
```

Первое, что следует отметить в этом тесте, – это использование параметра (done) в функции it.

Aurelia использует свойства асинхронного тестирования Jasmine всякий раз, когда мы применяем StageComponent. Поэтому мы должны не забывать передавать параметр done как часть нашей тестовой функции, а также вызывать функцию done после завершения теста.

Сам тест начинается с вызова функции create для экземпляра StageComponent (в котором находится тестовый экземпляр App), передавая функцию bootstrap. Функция create возвращает промис, к которому мы можем добавить функцию then, где можем определить фактическое содержание нашего теста. Как только мы откажемся внутри промиса, начальная загрузка и связывание уже будут выполнены. Затем мы можем запросить DOM через функцию document.querySelector. В этом тесте мы находим первый элемент <h1> и проверяем, чтобы визуализированный HTML-код содержал строку 'Please select :'.

Обратите внимание, что мы также создали функцию catch для промиса, который Aureila вернул при вызове create(bootstrap). Эта функция просто записывает сообщение об ошибке в консоль. Хотя это не является строго необходимым, но помогает улучшить ведение журнала ошибок при сбое тестов.

Прежде чем приступить к написанию дальнейших тестов, нам нужно обновить шаблон, который Aurelia использует при вызове функции inView:

```
.inView(`
<h1>` + '${Title}' + `</h1>
<ul>
   <div repeat.for="item of items"
      click.delegate="onItemClicked(item)">
      <button style="margin: 5px;" class="btn btn-primary">`
        + '${item.DisplayName}' + `</button>
   </div>
</ul>
<div>
   Selected Item : ` + '${SelectedItem.Id}
   - ${SelectedItem.DisplayName}' + `
</div>
<div class="form-group">
   <label for="inputName">Name :</label>
   <input type="text" class="form-control \
      input-name" id="inputName" value.bind="Name" />
</div>

<button id="submit-button-button" class="submit-button"
   click.delegate="onSubmitClicked()">Submit</button>
`)
```

Здесь мы просто скопировали полный файл `app.html` и включили его в вызов функции `inView`. Как отмечалось ранее, всякий раз при использовании подстановки параметров `${<propertyName>}` нам нужно будет переключаться на обычный синтаксис JavaScript с одинарными кавычками.

Наш следующий тест проверит, что массив кнопок был визуализирован в DOM:

```
it('should render buttons', function (done) {
   app.create(bootstrap).then(function () {
      var ulItemList = document.querySelectorAll(
         'ul > div > button');
      expect(ulItemList).toBeDefined();
      expect(ulItemList.length).toBe(3);
      for (var i = 0; i < ulItemList.length; i++) {
         var itemElement = ulItemList[i];
         expect(itemElement.innerHTML).toContain('Item');
      }
      done();
   }).catch(e => { console.log(`error : ${e}`); });
});
```

В этом тесте мы используем функцию `document.querySelectorAll` для возврата массива элементов кнопки. Обратите внимание на синтаксис CSS-селектора, который мы использовали: `ul > div > button`. Этот селектор вернет каждый элемент кнопки в элементе `div`, который является дочерним элементом элемента `ul`. Затем мы перебираем каждый возвращенный элемент и проверяем, что свойство innerHTML содержит слово '`Item`'. Помните, что текст кнопки, отображаемый на странице, – это `firstItem`, `secondItem` и `thirdItem`, поэтому каждая из этих кнопок будет содержать слово '`Item`'. Хотя этот тест может и не быть окончательным, он показывает, как мы можем использовать стандартные CSS-селекторы для возврата более чем одного дочернего элемента.

Теперь мы можем проверить правильность визуализации самой формы:

```
it('should render form with input element', function (done) {
   app.create(bootstrap).then(function () {
      let formGroup = document.querySelector('.form-group');
      let inputElement = formGroup.querySelector('input')
      expect(inputElement.value).toBe('Your Name');
      let submitButton = document
         .querySelector('#submit-button-button');
      expect(submitButton.innerHTML).toBe('Submit');
      done();
   }).catch(e => { console.log(`error : ${e}`);  });
});
```

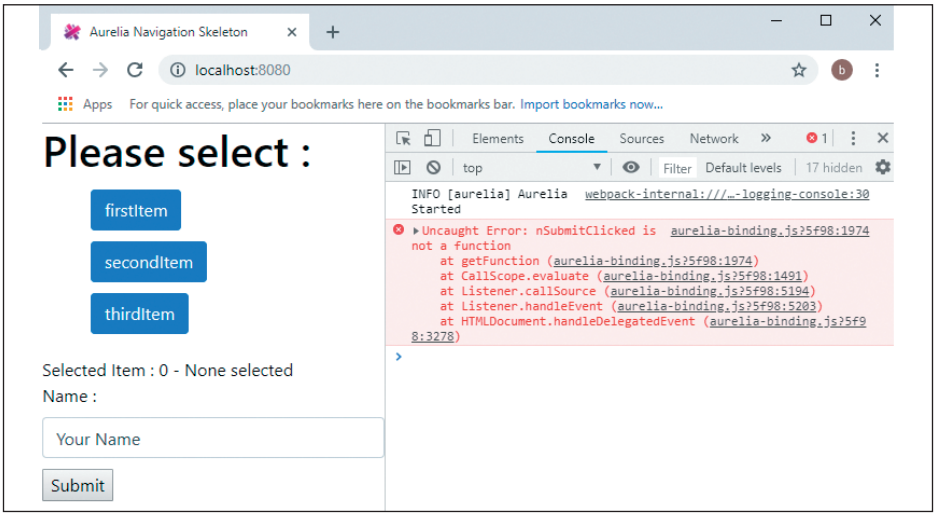
В этом тесте мы ищем в DOM элемент с CSS-классом `form-group`, который используется для идентификации `<div>` в нашей форме. Затем мы сохраняем этот элемент в переменной `formGroup`, после чего используем ее в качестве базы для поиска дочернего элемента ввода. В ходе теста мы проверяем, что значение входного элемента – `'Your Name'`, которое является значением по умолчанию, которое мы установили для формы. Последний элемент, который мы будем искать, – это кнопка ввода с атрибутом `'submit-button-button'`. Если внутреннее HTML-свойство содержит текст `'Submit'`, то наш тест пройдет. Обратите внимание, что мы могли бы проверить еще несколько CSS-классов или элементов нашей формы, но для краткости мы протестировали наиболее важные элементы.

События DOM

К сожалению, Aurelia не предоставляет механизм для тестирования событий DOM. Это означает, что мы не можем просто создать тест, который вводит данные в поле ввода и затем нажимает кнопку отправки формы, как мы это делали в случае с Backbone. Рассмотрим следующий фрагмент HTML-кода:

```
<button id="submit-button-button" class="submit-button"
  click.delegate="nSubmitClicked()">Submit</button>
```

Это фрагмент кода, который визуализирует нашу кнопку **Submit**. Мы уже видели тесты, которые удостоверяются, что эта кнопка видна на экране и что имя кнопки, по сути, – **Submit**. Но присмотритесь к названию функции `onSubmitClicked` для атрибута `click.delegate`. Оно содержит мелкую орфографическую ошибку: отсутствует начальная буква «o». Если мы скомпилируем эту версию кода, Aurelia не сообщит об этой ошибке на этапе сборки. Только во время выполнения, когда мы физически нажимаем на саму кнопку, мы получаем ошибку времени выполнения:



Ошибку, которую мы видим, очень просто воссоздать. Это означает, что мы не можем провести модульное тестирование результирующих взаимодействий DOM между нашим визуализированным HTML-кодом и нашими классами Aurelia. Отсутствие модульного тестирования означает, что нам нужно будет использовать сквозное тестирование с применением таких инструментов, как Protractor, чтобы найти ошибки такого рода в своем приложении, что намного дороже, чем просто написание модульного теста.

Резюмируя

На этом мы завершаем наш раздел о модульном и интеграционном тестированиях Aurelia. Мы видели, что Aurelia предоставляет опцию `au karma` для запуска Karma с целью модульного тестирования и что все настройки и зависимости автоматически устанавливаются за нас в ходе настройки командной строки Aurelia. Мы изучили, как создать компонент, как протестировать начальное состояние приложения и визуализацию элементов в DOM. К сожалению, нам не удалось протестировать сами события DOM, и нам потребуется выполнить интеграционное тестирование с помощью Protractor, чтобы протестировать приложение целиком.

Тестирование Angular

В этом разделе мы рассмотрим модульные тесты для нашего существующего приложения Angular, аналогичные тем, с которыми имели дело в Backbone и Aurelia.

Настройка

При создании проекта с использованием интерфейса командной строки Angular (с помощью команды ng new) в настройках проекта по умолчанию уже содержится весь стандартный код для запуска модульных тестов с использованием Karma и сквозных тестов с использованием Protractor. Настройка по умолчанию является очень удобной функцией Angular, сокращая время разработки конфигурации тестовой среды, и дает возможность погрузиться в написание тестов с самого начала проекта.

Чтобы запустить модульные тесты с использованием Karma, мы можем ввести в командной строке следующее:

```
ng test
```

Этот параметр командной строки будет компилировать и упаковывать наше приложение, а также запускать любые тесты, которые найдет в каталоге `src`. Любой файл TypeScript, имя которого совпадает с `*.spec.ts`, будет обозначен как тесто-

вый файл, и любые тесты в этом файле будут выполнены. К тому же при запуске Karma таким образом также автоматически будут отслеживаться изменения в наших файлах, перекомпилироваться и повторно запускаться наши тесты, так как были обнаружены изменения.

Проект по умолчанию в Angular создает файлы `.spec.ts` в исходном каталоге, рядом с тестируемыми компонентами. Это означает, что в каталоге `src/app` мы найдем и `app.component.ts`, и `app.component.spec.ts`. Хотя это и настройка по умолчанию, нет никаких причин, по которым мы не можем разбивать файлы `.spec.ts` на их собственные каталоги, если это необходимо. Однако наличие тестов в одном каталоге с тестируемым компонентом позволяет быстро открывать и изменять тесты, если изменяется сам компонент. Это также помогает определить, у каких компонентов нет модульных тестов.

Однако выполнение `ng test` в нашем проекте Angular на этом этапе приведет к ошибкам времени компиляции, а именно:

```
Can't bind to 'formGroup' since it isn't a known property of
'form'. ("
  <h2>Reactive Form :</h2>
  <form [ERROR ->][formGroup]="reactiveFormGroup"
(ngSubmit)="onSubmitRf()">
    <div class="form-group">
      <labe"): ng:///DynamicTestModule/AppComponent.html@28:6
  No provider for ControlContainer ("
```

Давайте посмотрим на существующий файл `app.component.spec.ts`, чтобы найти причину ошибки:

```
describe('/src/app/app.component.spec.ts: AppComponent ', () => {
  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [
        AppComponent
      ]
    }).compileComponents();
  }));
```

Здесь у нас есть функция `describe`, которая помечает начало нашего набора тестов, и функция `beforeEach`, которая устанавливает среду тестирования для нашего компонента. Обратите внимание, что эта функция помечена как `async`, что позволяет нам использовать синтаксис в стиле `async await`.

Первое, что делает этот тест, – вызывает функцию `TestBed.configureTestingModule` с объектом в качестве единственного аргумента. У этого объекта есть единственное свойство `declarations`, которое является массивом компонентов. Поскольку единственный компонент, который тестируется на данном

этапе, – это AppComponent, он единственный из перечисленных здесь. Функция configureTestingModule возвращает промис, поэтому мы можем использовать свободный синтаксис для вызова функции compileComponents после завершения этого вызова.

Функция configureTestingModule используется для установки любых зависимостей, которые могут понадобиться нашему компоненту. Эта функция, по сути, предоставляет нашему компоненту мини-среду Angular для работы. Это означает, что любые операторы импорта, используемые в нашем компоненте, должны будут иметь соответствующую запись в функции configureTestingModule.

Если мы посмотрим на файл app.component.ts, в верхней его части мы увидим, что импортируем три класса:

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

Здесь мы импортировали классы FormBuilder, FormGroup и FormControl из модуля '@angular/forms'. Этот оператор импорта является причиной ошибок компиляции. Поскольку наш тест полностью независим от запущенного приложения, он должен обеспечить закрытую среду для запуска компонента. Уловка для исправления этой ошибки состоит в том, чтобы импортировать модули, частью которых являются эти классы:

```
import { FormsModule, ReactiveFormsModule }
   from '@angular/forms';
describe('/src/app/app.component.spec.ts : AppComponent ', () => {
   beforeEach(async(() => {
      TestBed.configureTestingModule({
         declarations: [
            AppComponent
         ],
         imports: [
            FormsModule,
            ReactiveFormsModule
         ]
      }).compileComponents();
   }));
```

Здесь мы добавили оператор import в верхней части файла, чтобы импортировать FormsModule и ReactiveFormsModule из библиотеки '@angular/forms'. Мы также добавили свойство imports к объекту, переданному в функцию configureTestingModule, и перечислили в нем и FormsModule и ReactiveFormsModule.

Обратите внимание, что в Angular группы классов, интерфейсов и функций могут быть включены в так называемый модуль. В нашем примере AppComponent

нужна функция ngModel, которая является частью FormsModule, а также классы FormBuilder, FormGroup и FormControl, которые являются частью ReactiveFormsModule. Это означает, что при импорте FormsModule и ReactiveFormsModule в начале нашего теста все классы, интерфейсы и функции, предоставляемые этими модулями, доступны для нашей тестовой среды.

Внесение этого небольшого изменения в спецификацию теста позволит нам скомпилировать наши тесты. Далее мы рассмотрим создание тестов для нашего компонента Angular.

Тесты моделей

Для начала мы создадим серию тестов, которые будут проверять внутреннее состояние класса AppComponent после создания экземпляра в файле src/app/app.component.spec.ts:

```
describe('/src/app/app.component.spec.ts : AppComponent ', () => {
   let fixture: ComponentFixture<AppComponent>;
   let app: AppComponent;
   beforeEach(async(() => {
      TestBed.configureTestingModule({
         declarations: [
            AppComponent
         ],
         imports: [
            FormsModule,
            ReactiveFormsModule
         ]
      }).compileComponents();
      fixture = TestBed.createComponent(AppComponent);
      app = fixture.debugElement.componentInstance;
   }));
});
```

Мы определили наш набор тестов, используя функцию beforeEach, и начали с определения двух переменных, fixture и app. Переменная fixture имеет тип ComponentFixture<AppComponent> и является результатом вызова TestBed.createComponent(AppComponent). Она содержит представление AppComponent в DOM и может использоваться для запроса DOM или для непосредственного управления элементами DOM. Мы будем использовать свойство fixture позже, когда начнем взаимодействовать с элементами DOM.

Переменная app является экземпляром самого класса AppComponent и доступна через свойство debugElement.componentInstance переменной fixture. Получив дескриптор экземпляра класса AppComponent, мы можем начать тестировать, что каждое из его свойств было инициализировано правильно:

```
it('should create the app', () => {
  expect(app).toBeTruthy();
});

it('should set the Title property', () => {
  expect(app.Title).toBe('Please select :');
});

it('should set the SelectedItem property', () => {
  expect(app.SelectedItem.Id).toBe(0);
  expect(app.SelectedItem.DisplayName).toBe('None selected');
});

it('should set the items property to an array', () => {
  expect(app.items.length).toBe(3);
  for (let item of app.items) {
    expect(item.DisplayName).toContain('Item');
  }
});
```

Здесь у нас четыре теста. Первый тест просто гарантирует, что переменная app была создана правильно. Хотя этот тест может показаться тривиальным, он должен быть первым тестом для любого нового компонента Angular и фактически первым тестом, создаваемым интерфейсом командной строки Angular. Данный тест очень полезен при запуске тестирования компонентов, так как он вынудит фреймворк гарантировать, что все зависимости компонентов были импортированы правильно. Вспомните, что нам нужно было изменить функцию `beforeEach` и убедиться, что `FormsModule` и `ReactiveFormsModule` были импортированы правильно до компиляции тестов? Что ж, этот простой тест действительно существует для того, чтобы гарантировать, что тесты компонента могут компилироваться и что зависимости, которые нужны компоненту, были инициализированы правильно.

Следующие три теста гарантируют, что свойства `Title`, `SelectedItem` и `items` класса `AppComponent` были созданы правильно. Эти тесты гарантируют, что внутреннее состояние класса является правильным во время создания.

Тесты визуализации

Теперь мы можем обратить наше внимание на сгенерированный HTML-код, который создает наше приложение, и проверить, правильно ли наш компонент визуализировал элементы в DOM:

```
it("should render 0 - none selected to the DOM", () => {
  fixture.detectChanges();
  fixture.whenStable().then(() => {
    const domElement = fixture.debugElement.nativeElement;
```

```
      let selectedItemDiv =
        domElement.querySelector("#selectedItemText");
      expect(selectedItemDiv).toBeTruthy();
      expect(selectedItemDiv.innerHTML).
        toContain('0 - None selected');
    });
});
```

Здесь мы определили тест, который проверит, что текст `0 - none selected` был визуализирован в DOM. Тест начинается с вызова функции `fixture.detectChanges`. Эта функция предоставляется фреймворком Angular и запускает цикл обнаружения изменений для компонента. По сути, функция `detectChanges` создаст компонент в DOM. Затем мы присоединяемся к промису `fixture.whenStable`, используя функцию `then`, которая будет ожидать, пока компонент завершит визуализацию в DOM, прежде чем вызвать наш промис. Эти два вызова, `Detechanges` и `whenStable`, всегда используются парно в тестах Angular.

Обратите внимание, что функция `deteChanges` создаст экземпляр нашего компонента. Это означает, что если нам нужно создать шпионов для какой-либо из функций в нашем компоненте, эти шпионы должны будут быть созданы до вызова функции `detectChanges`.

Затем наш тест создает переменную `domElement`, чтобы обратиться к `debugElement.nativeElement` самой фикстуры. Это собственный DOM-элемент компонента, который используется для поиска DOM. После этого мы создаем переменную `selectedItemDiv`, которая является результатом использования функции `querySelector` для ссылки `domElement`. Функция `querySelector` использует стандартные шаблоны поиска jQuery, чтобы найти элементы, и в этом случае она выполняет поиск элемента DOM с атрибутом `selectedItemText`. Наш первый оператор `expect` проверяет, был ли элемент фактически найден, а последний оператор проверяет свойство `innerHTML` нашего элемента DOM, чтобы увидеть, содержит ли он текст `0 - none selected`.

Следовательно, наш основной цикл тестирования для Angular выглядит так:

- используйте функцию `TestBed.configureTestingModule`, чтобы установить любые зависимости для нашего тестируемого компонента;
- скомпилируйте компонент, применяя функцию `compileComponents`;
- вызовите функцию `TestBed.createComponent`, чтобы настроить среду модульного тестирования для нашего компонента;
- вызовите функцию `detectChanges`, чтобы запустить цикл обнаружения изменений;
- присоединитесь к промису `whenStable`, чтобы дождаться, пока наш компонент будет визуализирован в DOM;
- используйте свойство фикстуры `debugElement.nativeElement`, чтобы получить дескриптор к визуализированному DOM;

- используйте функцию querySelector для запроса DOM с использованием синтаксиса jQuery.

Используя этот шаблон тестирования, мы теперь можем проверить оставшиеся элементы DOM:

```
it("should render 3 buttons to the DOM", () => {
   fixture.detectChanges();
   fixture.whenStable().then(() => {
      const domElement = fixture.debugElement.nativeElement;
      let selectedItemDiv = domElement.querySelector('ul');
      let buttons = selectedItemDiv.querySelectorAll('div > button');
      expect(buttons.length).toBe(3);
      for (let button of buttons) {
         expect(button.innerHTML).toContain('Item');
      }
   });
});
```

Здесь мы написали тест, который выполняет проверку на предмет наличия трех кнопок из нашей коллекции. Наш тест начинается с поиска первого элемента <ul> в DOM, а затем использует функцию querySelectorAll, чтобы найти все элементы <button> в этом элементе <ul>. Функция querySelectorAll будет возвращать массив элементов, где функция querySelector будет возвращать только первый соответствующий элемент. Затем наш тест перебирает каждую из этих кнопок и проверяет, что каждая кнопка содержит текст 'Item'.

Наш следующий тест проверит, что наша первая форма была визуализирована правильно:

```
it("should render a boostrap form-group to the DOM", () => {
   fixture.detectChanges();
   fixture.whenStable().then(() => {
      const domElement = fixture.debugElement.nativeElement;
      let selectedItemDiv = domElement.querySelector(".form-group");
      let formGroup = domElement.querySelector(".form-group input");
      expect(formGroup.value).toBe('Your Name');
   });
});
```

Здесь мы находим элемент DOM с классом form-group, используя селектор запросов ".form-group", а внутри этого элемента – первый элемент input. Мы ожидаем, что значение этого элемента будет установлено правильно при инициализации формы.

Финальный тест визуализации для второй реактивной формы выглядит так:

```
it("should render an Angular formGroup to the DOM", () => {
   fixture.detectChanges();
   fixture.whenStable().then(() => {
      const domElement = fixture.debugElement.nativeElement;
      let formGroup = domElement.querySelector("form input");
      expect(formGroup.value).toBe('RF Input');
   });
});
```

Здесь мы находим DOM-элемент `form`, а внутри него – первый элемент `input`. Мы ожидаем, что значение этого элемента будет установлено правильно при инициализации формы.

На этом наши тесты визуализации DOM в Angular окончены.

Тесты форм

Последний набор тестов, который мы создадим, будет нацелен на две наши формы и будет имитировать ввод пользователем значений в поля формы, а также нажатие кнопки **Отправить**. Первый тест предназначен для нашей стандартной формы:

```
it("should set a value on the form, and click submit", () => {
   let submitSpy = spyOn(app, 'onSubmit');
   fixture.detectChanges();
   fixture.whenStable().then(() => {
      const domElement = fixture.debugElement.nativeElement;
      let formInput = domElement.querySelector(".form-group
input");
      expect(formInput.value).toBe(`Your Name`);
      formInput.value = 'Updated Value';
      let submitButton =
         domElement.querySelector('#submit-button-button');
      expect(submitButton).toBeTruthy();
      submitButton.click();
      expect(submitSpy).toHaveBeenCalled();
   });
});
```

Здесь у нас есть тест, который начинается с создания шпиона `submitSpy` для функции нашего компонента, `onSubmit`. Помните, что при вызове `DetectChanges` будет фактически создан наш компонент и прикреплен к DOM, поэтому любой шпион, которого мы создаем, должен быть до вызова `Detechanges`. Наш тест начинается с того же кода, который мы использовали в предыдущем тесте визуализации DOM, поскольку мы находим элемент `input` и ожидаем, что его начальное свойство `value` установлено правильно. Затем мы устанавливаем свойство `value` переменной `formInput`. Этот вызов имитирует ввод пользователем зна-

чения в элемент input. После этого мы находим кнопку submit, используя функцию querySelector, как обычно, и вызываем функцию click, что будет имитировать пользователя, нажимающего на кнопку **Отправить**. Затем тест ожидает, что наш submitSpy был вызван.

Таким образом, используя тесты форм подобного рода, мы можем взаимодействовать с нашей HTML-формой и моделировать действия пользователя в рамках модульного теста.

Нам также нужно будет создать несколько тестов для нашей реактивной формы:

```
it("should set a value on the reactive form, and click submit",
   () => {
   let submitSpy = spyOn(app, 'onSubmitRf');
   fixture.detectChanges();
   fixture.whenStable().then(() => {
      const formGroup = app.reactiveFormGroup;
      expect(formGroup).toBeTruthy();
      expect(formGroup.value.nameInput).toBe('RF Input');
      formGroup.controls[‘nameInput’].setValue('Updated RF Value');
      const domElement = fixture.debugElement.nativeElement;
      let submitButton =
         domElement.querySelector('#rf-submit-button');
      expect(submitButton).toBeTruthy();
      submitButton.click();
      expect(submitSpy).toHaveBeenCalled();
   });
});
```

В этом тесте есть ряд интересных настроек для работы с реактивными формами Angular. Чтобы получить или установить значения из реактивной формы, нам нужно будет напрямую обратиться к FormGroup внутри компонента. В нашем тесте мы создаем переменную formGroup, чтобы напрямую обратиться к свойству reactiveFormGroup в экземпляре app. Всегда полезно проверить, что экземпляр FormGroup был создан правильно, так как это может произойти в жизненном цикле страницы позже, чем мы ожидаем. По этой причине мы проверяем, что переменная formGroup сама возвращает истинное значение. Получив ссылку на экземпляр FormGroup, мы можем проверить правильность установки значения nameInput с помощью свойства formGroup.value.nameInput.

Помните, что когда мы устанавливали начальное значение FormGroup, то использовали функцию reset:

```
this.reactiveFormGroup.reset({
   nameInput: 'RF Input'
});
```

Внутри `FormGroup` это делается для создания простого Java-объекта POJO с единственным свойством `nameInput`. Любое из этих имен свойств может быть доступно через свойство `value` в `FormGroup`, что дает нам доступ к этому свойству путем проверки `formGroup.value.nameInput` или для полного имени `app.reactiveFormGroup.value.nameInput`.

Следующая строка нашего теста устанавливает значение этого элемента управления формы:

```
formGroup.controls['nameInput'].setValue('Updated RF Value');
```

Опять же, когда мы создаем реактивную форму, сам элемент `FormGroup` создается следующим образом:

```
this.reactiveFormGroup = this.formBuilder.group({
   nameInput: new FormControl({})
});
```

Итак, сам `FormGroup` создается с объектом, который имеет набор свойств, и каждое из этих свойств является экземпляром `FormControl`. Поэтому объект `FormGroup` разрешает доступ к каждому из этих элементов управления через свойство `controls`. Затем мы можем обратиться к индивидуальному элементу управления по имени. Когда у нас есть дескриптор необходимого нам `FormControl`, мы можем вызвать функцию `setValue` для имитации обновления пользователем значения в нашей форме. После того как значение формы было установлено, мы можем запросить DOM о правильной кнопке **Submit** и щелкнуть на нее, чтобы вызвать отправку формы. Затем мы ожидаем, что функция `onSubmitRf` будет вызвана с помощью нашего шпиона.

Резюмируя

Angular включает в себя все необходимые библиотеки и конфигурацию, чтобы делать возможным как модульное тестирование с Karma, так и приемочное тестирование через Protractor из коробки при работе с новым проектом.

Этот фреймворк был построен с твердым вниманием к тестированию, поэтому он обладает полным жизненным циклом компонентов, который мы можем использовать в нашем коде при тестировании. Мы можем протестировать каждый элемент нашего кода, включая тесты визуализации DOM и взаимодействия форм. В то время как для настройки теста через классы `ComponentFixture` может потребоваться некоторое время, чтобы разобраться и выполнить конфигурацию, это отличный фреймворк, который дает все, что потребуется компоненту для работы в изолированной среде.

Тестирование с React

В последнем разделе этой главы мы создадим набор модульных тестов для нашего приложения в React. Подобно тому, что мы сделали с другими фреймворками, мы проверим правильность начального состояния нашего приложения при запуске, а затем поработаем с тестами визуализации DOM. Наконец, мы создадим набор тестов, которые будут заполнять значения формы, а затем отправлять форму.

Несколько точек входа

В нашем приложении React используется Webpack в качестве инструмента компиляции и связывания для преобразования наших файлов TypeScript в пригодные для использования компоненты React. В процессе пакетирования Webpack нам нужно указать точку входа нашего приложения, а также имя файла на выходе. Поэтому, учитывая точку входа `/app/index.tsx` и имя файла `/dist/bundle.js`, все наши файлы кода TypeScript окажутся в файле `bundle.js`.

Все это хорошо, но при создании тестов нашей точкой входа является не само приложение, а спецификация теста. Это означает, что нам нужно настроить Webpack для генерации разных пакетов на основе разных точек входа. Это можно сделать довольно просто, обновив файл `webpack.conf.js`:

```
const config = {
   entry: {
      app: './src/index.tsx',
      test: './test/react.app.tests.tsx'
      },
   output: {
      path: path.resolve(__dirname, 'dist'),
      filename: '[name].js'
   },
```

Здесь мы изменили свойство entry из файла (`entry: "./app/index.tsx"`), чтобы включить второй файл записи с именем `test`. Это позволяет нашему браузерному приложению использовать один файл `.tsx` в качестве точки входа, а нашему тестовому приложению использовать другой файл .tsx в качестве точки входа.

Второе изменение, которое мы сделали, заключается в использовании `[name]` точки входа в качестве вывода процесса объединения Webpack. Это означает, что мы получим два пакета, `app.js`, с точкой входа `/app/index.tsx` и `test.js` с точкой входа `/tests/react.app.tests.tsx`. Webpack скомпилирует приложение и сгенерирует эти пакеты за нас в каталоге `dist`.

Использование Jest

React предлагает использовать Jest в качестве библиотеки для модульного тестирования по умолчанию. Все наши предыдущие образцы использовали Karma, но интеграция между React и Jest означает, что его очень просто настроить, и мы можем остаточно быстро запускать свои тесты. В этом разделе мы будем использовать Jest для наших тестов React. Чтобы установить Jest, просто введите следующее:

```
npm install jest -save-dev
```

Наряду с Jest нам понадобится Jasmine и его типы:

```
npm install jasmine @types/jasmine --save-dev
```

После установки нам нужно внести небольшое изменение в файл `package.json`, чтобы запустить Jest и свои тесты:

```
{
   "name": "react-sample",
   "version": "1.0.0",
   "description": "",
   "main": "index.js",
   "scripts": {
   "test": "jest"
},
...существующие свойства
```

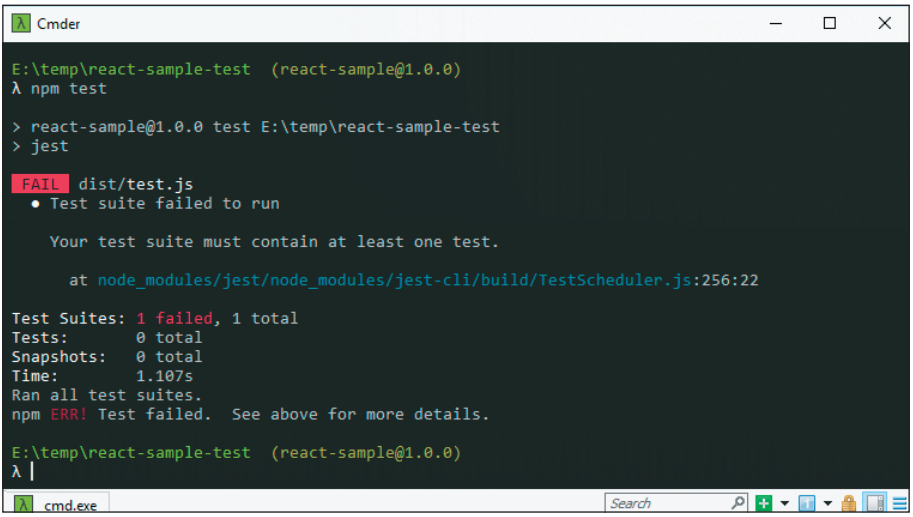
Здесь мы изменили свойство scripts и заменили существующее свойство `test` словом `"jest"`. Это означает, что мы можем запустить npm из командной строки и использовать аргумент `test`, чтобы запустить процесс npm, который запустит jest. Помните, что Jest устанавливается как зависимость проекта и не устанавливается глобально, поэтому простой запуск `jest` из командной строки не сработает.

Создав запись сценария `npm` с именем `test`, мы можем использовать среду `npm`, которая устанавливается локально, и `npm` сможет найти исполняемый файл `jest` в каталоге `node_modules`.

Поэтому мы можем запустить Jest из командной строки:

```
npm test
```

К сожалению, мы еще не написали никаких тестов, поэтому Jest ответит неудачным выполнением теста:



Начнем с написания набора тестов, которые проверяют внутреннее состояние приложения при запуске. Помните, что у нашего приложения React есть два свойства, которые установлены как свойства по умолчанию. Это свойства `title` и `SelectedItem`. Тест свойства `title` будет выглядеть так:

```
import * as React from "react";
import * as ReactDOM from "react-dom";
import * as ReactTestUtils from "react-dom/test-utils";
import { ItemCollectionView, ClickableItemArray }
   from '../src/ReactApp';

describe('/test/react.app.tests.tsx: ArrayView tests', () => {
   let renderer: any;
   beforeEach(() => {
      renderer = ReactTestUtils.renderIntoDocument(
         <ItemCollectionView items={ClickableItemArray} />
      );
   });
   it('should render Title property', () => {
      let domNode = ReactDOM.findDOMNode(renderer) as Element;
      let title = domNode.querySelector("h1") as Element;
      expect(title.textContent).toBe('Please select:');
   });
});
```

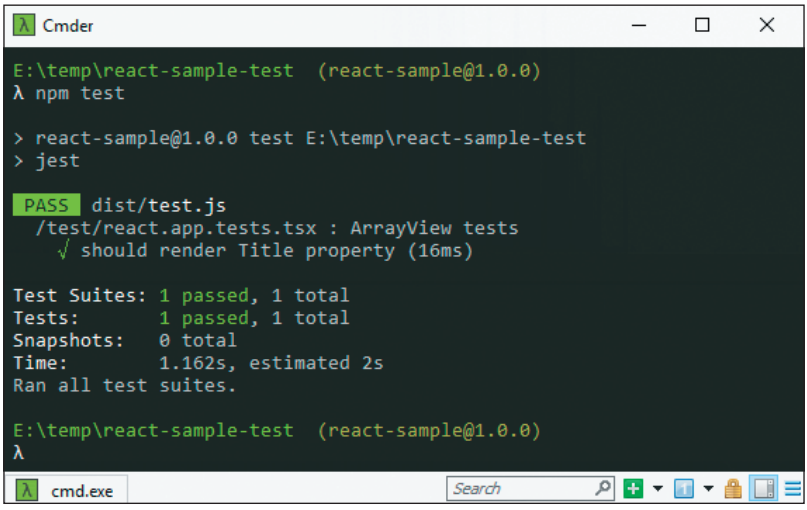
Структура этого тестового файла должна быть уже достаточно знакома. Вначале идет несколько операторов `import` для загрузки библиотек: `React`, `ReactDOM` и `ReactTestUtils`. Затем мы загружаем компонент `ItemCollectionView` React, а также массив `ClickableItem` из файла `src/ReactApp.tsx`. Наш тест

использует Jasmine, поэтому здесь есть уже знакомые нам функции `describe`, `beforeEach` и `it`.

Наш тест устанавливает переменную `renderer`, которая присваивается в рамках функции `beforeEach`. Мы используем функцию `renderIntoDocument` статического экземпляра `ReactTestUtils` для создания тестового экземпляра компонента `ItemCollectionView`. Обратите внимание, что функция `renderIntoDocument` принимает фрагмент HTML-кода в стиле React, который очень похож на фрагмент в файле `index.tsx`. Он использует синтаксис React для настройки тестовой среды для компонента `ItemCollectionView`. Это очень похоже на другие фреймворки, которые создают определенную среду тестирования для тестируемого объекта. У `ReactTestUtils` есть ряд функций, доступных для создания и управления средой для нашего тестируемого объекта.

Тест начинается с создания переменной `domNode`, которая будет содержать самый верхний элемент DOM нашего компонента. Затем мы используем функцию `ReactDom.findDOMNode` и передаем переменную `renderer` в качестве аргумента. Эта функция вернет дескриптор на самый верхний элемент DOM, созданный для нашего компонента. Получив этот дескриптор, мы можем использовать его для вызова стандартной функции `querySelector` или `querySelectorAll` для поиска элементов в DOM. Наш тест использует функцию `querySelector`, чтобы найти первый элемент `<h1>`, а затем проверяет, что для `textContent` этого элемента установлено значение `'Please select:'`.

Это процесс, которому мы будем следовать для большинства наших тестов React. Мы используем экземпляр, возвращенный вызовом функции `renderIntoDocument`, чтобы найти самый верхний узел DOM для своего компонента, а затем используем этот узел для запроса каждого элемента, который мы ищем. После запуска `npm test` этот тест теперь будет выполнен, и мы получим сообщение о прохождении:



Тесты начального состояния

Как упоминалось ранее, наше приложение React запускается с начальным состоянием. Свойства `title` и `SelectedItem` установлены, и приложение должно отобразить на экране три кнопки. Давайте теперь расширим наш набор тестов для проверки этих элементов. Во-первых, мы будем проверять свойство `SelectedItem` на предмет правильной визуализации в DOM:

```
it('should render SelectedItem property', () => {
   let domNode = ReactDOM.findDOMNode(renderer) as Element;
   let selectedItem = domNode.querySelector("#selectedItem")
      as Element;
   expect(selectedItem.innerHTML).toContain("0 - None Selected");
});
```

Здесь мы использовали тот же шаблон, что и в нашем предыдущем тесте, чтобы найти элемент DOM с идентификатором `selectedItem`. Затем мы проверяем, что элемент `innerHTML` визуализируется правильно. В этом тесте нет ничего особенного, кроме того факта, что мы используем свойство DOM-узла `innerHTML` вместо свойства `textContent`, которое использовалось ранее. Теперь давайте проверим, что кнопки были визуализированы правильно:

```
it('should find three buttons', () => {
   let domNode = ReactDOM.findDOMNode(renderer) as Element;
   let ulElement = domNode.querySelector("ul") as Element;
   let buttons = ulElement.querySelectorAll("button")
      as NodeListOf<HTMLButtonElement>;

   expect(buttons.length).toBe(3);
   for (let i = 0; i < buttons.length; i++) {
      expect(buttons[i].textContent).toContain("Item");
   }
});
```

В этом тесте есть несколько интересных моментов. Мы начинаем с получения дескриптора узла DOM посредством вызова `ReactDOM.findDOMNode`, как обычно. Затем мы используем этот дескриптор, чтобы найти первый элемент `<ul>`, который мы храним в переменной `ulElement`. Обратите внимание, что потом мы применяем дескриптор `ulElement` для повторного вызова `querySelectorAll`, чтобы найти все дочерние элементы кнопки. Это означает, что мы можем направлять поиск элементов в DOM, находя элемент верхнего уровня, а затем используя этот элемент в дальнейших запросах, чтобы уточнить поиск.

При поиске кнопок внутри элемента `<ul>` верхнего уровня применяется функция `querySelectorAll`, так как мы ожидаем, что будет найден массив элементов кнопки. Второй интересный момент в этом поиске кнопок в DOM – тип, ко-

торый возвращает функция `querySelectorAll`. Мы приводим тип возвращаемого значения к типу `NodeListOf<HTMLButtonElement>`. Это означает, что мы ожидаем, что вызов функции `querySelectorAll` вернет список узлов DOM типа `HTMLButtonElement`. Файлы определений TypeScript для React содержат исчерпывающий список элементов, которые могут быть возвращены при вызове функций `querySelector` и `querySelectorAll`.

Если щелкнуть правой кнопкой мыши функцию `querySelector` и выбрать **Go to Definition** (**Перейти к определению**), мы увидим определение этой функции:

```
querySelector<K extends keyof HTMLElementTagNameMap>
  (selectors: K): HTMLElementTagNameMap[K] | null;
```

Данное определение функции сообщает нам следующую информацию:

Обобщенный параметр, который передается с именем `K`, использует ключевое слово `keyof`, чтобы ограничить тип `K` и разрешить только значения, являющиеся свойствами типа `HTMLElementTagNameMap`. Если мы снова щелкнем правой кнопкой мыши по типу `HTMLElementTagNameMap`, то увидим, что это интерфейс, который определяет все возможные комбинации типа `K` и их возвращаемые типы:

```
interface HTMLElementTagNameMap {
  "a": HTMLAnchorElement;
  "abbr": HTMLElement;
  "address": HTMLElement;
  "applet": HTMLAppletElement;
  "area": HTMLAreaElement;
  ...
}
```

Оператор `keyof` при применении к этому интерфейсу вернет строковый литерал, который должен быть равен либо `"a"`, либо `"abbr"`, либо `"address"`, либо `"applet"` и т. д. Это означает, что мы можем вызывать функцию `querySelector` только с определенным набором строк, где каждая строка отображается в определенный набор HTML-типов, которые мы можем использовать. Каждое из этих строковых значений, которые генерируются с помощью ключевого слова `keyof`, будет возвращать элемент определенного типа. В нашем случае строковый литерал `"button"` вернет тип элемента `HTMLButton`. Это пример очень хорошего и практического использования ключевого слова `keyof`, которое позволяет ограничивать аргументы вызова функции определенным набором строк и где каждое строковое значение будет возвращать разный тип.

Теперь, когда у нас есть список наших кнопок, мы можем проверить, что их поля `textContent` были установлены правильно.

Элемент input

Давайте теперь напишем набор тестов для нашей формы React. Первый тест, который мы напишем, будет проверять, что элемент input был установлен в правильное начальное значение. Опять же, этот тест важен в жизненном цикле форм, поскольку в реальном приложении мы могли бы установить эти начальные значения из существующей записи базы данных. Наш тест выглядит так:

```
it('should render a form with default value ', () => {
   let domNode = ReactDOM.findDOMNode(renderer) as Element;
   let form = domNode.querySelector("form") as Element;
   expect(form).toBeTruthy();

   let label = form.querySelector("label")!;
   expect(label.innerHTML).toContain("Name :");

   let input = form.querySelector("input")!;
   expect(input.value).toBe('Your Name');
});
```

Наш тест начинается с поиска элемента form в визуализированном DOM, а затем проверяется, что этот элемент действительно был найден. Затем тест находит метку для нашего элемента ввода и гарантирует, что для него установлено значение "Name :". Однако еще раз взглянем на функцию `querySelector` для элемента `label`. Обратите внимание, что здесь мы используем оператор ненулевого утверждения TypeScript (`!`), чтобы отметить тип, который возвращает `querySelector`. Это сделано, чтобы сообщить компилятору, что мы уверены, что возвращаемое значение не будет нулевым или неопределенным. Без этого оператора TypeScript будет выдавать многочисленные ошибки компиляции, утверждая, что переменная может быть нулевой.

В предыдущей главе при обсуждении оператора ненулевого утверждения автор указал, что в стандартном коде не должно быть случаев, когда этот оператор следует использовать. Однако сейчас, кажется, подходящий случай для этого. Давайте посмотрим, почему. В нашем тесте имеется утверждение непосредственно после попытки найти элемент DOM. В этом случае нам не нужно обороняться, чтобы иметь дело с переменной, если она может быть нулевой. Наш тест не пройдёт, если элемент не может быть найден или если у элемента нет правильного текста в свойстве `innerHTML`. Поэтому мы можем указать компилятору TypeScript, что нас устраивает элемент, равный нулю, так как наш тест все равно завершится неудачей, если он будет. В этом случае оператор ненулевого утверждения избавляет нас от чрезмерной сложности наших тестов.

Наш тест продолжается, находя элемент `input` в DOM, а затем проверяет атрибут `value` этого элемента, чтобы убедиться, что для него установлено значение "Your Name".

Помните, что HTML-синтаксис для установки значения начального элемента `input` выглядит так:

```
<input type="text" value="Your Name"/>
```

Таким образом, `value` является атрибутом элемента `input` и должно быть установлено при визуализации формы для предварительного заполнения элемента.

Давайте теперь посмотрим, как обновить значение формы:

```
it('should update form value', () => {
   let domNode = ReactDOM.findDOMNode(renderer) as Element;
   let form = domNode.querySelector("form") as Element;
   expect(form).toBeTruthy();

   let input = form.querySelector("input")!;
   input.value = 'pdatedInputValue';

   ReactTestUtils.Simulate.change(input);

   expect(renderer.state.inputName).toBe('updatedInputValue');
});
```

Здесь у нас есть тест, который находит элемент `form`, а затем дочерний элемент `input`.

Затем мы устанавливаем значение элемента `input` в `updatedInputValue`, просто устанавливая свойство `value`. После этого мы вызываем функцию `ReactTestUtils.Simulate.change`, передавая элемент, который хотим обновить. Функция `Simulate` отправит событие DOM с необязательными данными в элемент управления. Таким образом мы обновляем элемент DOM, аналогично функции `Angular`, `detectChanges`. Как только это будет сделано, мы можем проверить свойство `state` переменной `renderer`, чтобы убедиться, что свойство `inputName` обновлено правильно.

Помните, что React использует свойство `state` для получения и установки значений, с которыми работает наш компонент. Как только пользователь изменил значение в форме, поскольку это значение привязано к свойству компонента React, свойство `state` будет содержать обновленное значение.

Отправка формы

Наш последний набор тестов для форм React будет представлять саму форму. Помните, что мы объявили функцию, которая будет вызываться при отправке формы:

```
<form onSubmit={this.onSubmit} >
```

Здесь наш компонент React указал, что функция onSubmit должна вызываться при отправке самой формы. Затем эту функцию можно использовать для преобразования значений формы в структуру JSON для отправки в конечную точку REST. Хотя здесь мы не будем показывать, как взаимодействовать с конечной точкой REST, важным шагом для наших тестов является обеспечение вызова функции onSubmit.

При использовании React на самом деле есть два способа, с помощью которых можно смоделировать событие отправки формы. Давайте посмотрим на первый способ:

```
it('should trigger onSubmit when form is submitted', () => {
   let spy = spyOn(ItemCollectionView.prototype, 'onSubmit');
   let testRenderer = ReactTestUtils.renderIntoDocument(
      <ItemCollectionView items={ClickableItemArray} /> ) as any;
   let formForSubmit = ReactTestUtils.
      findRenderedDOMComponentWithTag (testRenderer, 'form');
   ReactTestUtils.Simulate.submit(formForSubmit);
   expect(spy).toHaveBeenCalled();
});
```

Здесь у нас есть тест, который начинается с создания шпиона для функции onSubmit нашего компонента React. Опять же, нам нужно создать этого шпиона до того, как компонент будет визуализирован в DOM, поэтому мы присоединяем шпиона к определению класса ItemCollectionView с помощью свойства ItemCollectionView.prototype. Затем мы создаем переменную testRenderer и вызываем функцию ReactTestUtils.renderIntoDocument, чтобы создать компонент ItemCollectionView и визуализировать его в DOM.

Далее мы используем функцию ReactTestUtils.findRenderedDOMComponent WithTag, чтобы найти экземпляр нашей формы. Эта функция похожа на наши предыдущие функции поиска в DOM. Получив дескриптор формы, мы можем вызвать функцию ReactTestUtils.Simulate.submit для отправки формы. После этого наш тест проверяет, что шпион был вызван, или, другими словами, что для нашего компонента была вызвана функция onSubmit.

Обратите внимание, что хотя функция ReactTestUtils.Simulate.submit и будет отправлять форму за нас, существует другой способ инициировать отправку формы. В нашей форме также есть кнопка Submit, которая выглядит так:

```
<button className="submit-button" type="submit"
   value="Submit">Submit</button>
```

Эта кнопка относится к типу "submit", и, нажав на нее, вы активируете отправку формы. Это означает, что более точным тестом для отправки формы будет инициирование события click и обеспечение правильной отправки формы. В этом случае наш тест будет выглядеть так:

```
it('should trigger onSubmit when button is clicked', () => {
   let spy = spyOn(ItemCollectionView.prototype, 'onSubmit');
   let testRenderer = ReactTestUtils.renderIntoDocument(
      <ItemCollectionView items={ClickableItemArray} /> ) as any;
   let button = ReactTestUtils.scryRenderedDOMComponentsWithTag
      (testRenderer, 'button');
   ReactTestUtils.Simulate.submit(button[3]);
   console.log('after calling submit form');
   expect(spy).toHaveBeenCalled();
});
```

Наш тест начинается с создания шпиона и визуализации компонента React в DOM, как обычно. Однако обратите внимание, что мы используем другую функцию тестирования, `scryRenderedDOMComponentsWithTag`. Эта функция вернет все элементы, которые она найдет в DOM, соответствующие имени тега. Мы ищем элементы типа `button` и поэтому в действительности найдем четыре элемента в нашем компоненте. Первые три кнопки отображаются как выбираемые элементы, а четвертая кнопка – это кнопка отправки формы. Поэтому наш тест вызывает функцию `ReactTestUtils.Simulate.submit` с четвертой кнопкой для отправки формы. После этого мы ожидаем, что шпион был вызван, а это означает, что функция `onSubmi` была запущена при отправке формы.

Подводя итоги

React имеет диапазон тестирования, аналогичный другим рассмотренным нами средам, и позволяет создавать компонент в DOM, специально настроенном для тестируемого компонента. Мы можем запросить этот DOM, чтобы найти каждый из искомых элементов, и гарантировать, что они отображаются правильно. Затем мы исследовали методы, которые React использует для установки значений формы при ее создании и извлечения этих значений после их обновления. Подобно `setFixtures` в Jasmine и `TestBed` в Angular, React предоставляет библиотеку `ReactTestUtils` для помощи при визуализации компонентов в DOM и имитации событий. React использует Jest в качестве библиотеки для модульного тестирования вместо Karma.

Резюме

В этой главе мы рассмотрели немало вопросов. Мы подробно рассмотрели, как проводить модульное тестирование каждого нашего фреймворка, совместимого с TypeScript. Все фреймворки, которые мы использовали, позволяли создавать компоненты в тестовом DOM, предназначенном для тестируемого компонента. Мы смогли запросить элементы DOM, убедиться, что они были правильно инициализированы, и сымитировали пользователя, вводящего значения в форму и нажимающего кнопку **Отправить**. Все наши фреймворки предоставляли ана-

логичный набор функций, за исключением Aurelia, который не позволил нам тестировать события DOM, такие как нажатия кнопок.

Модульное тестирование – это действительно образ мышления. Некоторые разработчики и даже некоторые команды разработчиков могут думать о компонентах с точки зрения тестируемости и пишут модульные тесты во время разработки компонентов. Некоторые разработчики предпочитают сначала создать целый компонент, а затем набор модульных тестов для обеспечения функциональности. Какой бы путь ни выбрала ваша команда, помните, что без модульных тестов мы действительно просто случайным образом меняем вещи каждый раз, когда модифицируем код. Модульные тесты дают нашему коду возможность полностью измениться внутренне, без риска нарушения какой-либо функциональности. Модульные тесты дают нашим приложениям знак качества, гарантирующий их правильную работу в любое время.

В нашей следующей главе мы подробно рассмотрим модуляризацию, используя CommonJ (вместе с Node) и загрузку модулей в стиле AMD (вместе с Require). Мы также рассмотрим метод загрузки модулей SystemJS, который объединяет в себе как CommonJ, так и AMD.

Глава 10

Модуляризация

Модуляризация – это популярный метод, используемый в современных языках программирования, который позволяет создавать программы из ряда небольших библиотек или модулей. Написание программ, использующих модули, побуждает программистов писать код, соответствующий принципу проектирования под названием **разделение ответственности**. Основной принцип разделения ответственности заключается в том, что мы должны программировать с использованием определенного интерфейса. Это означает, что код, реализующий этот интерфейс, может быть реорганизован, улучшен или даже полностью заменен, не затрагивая остальной части программы. Это также помогает при тестировании нашего кода, поскольку код, обеспечивающий реализацию интерфейса, может быть легко заблокирован или смоделирован в тестовом сценарии.

В JavaScript до ECMAScript 6 не было концепции модулей. Популярные фреймворки и библиотеки, такие как **Node** и **Require**, реализовали свои собственные библиотеки синтаксиса загрузки модулей, чтобы заполнить этот пробел. К сожалению, сообществом JavaScript были приняты два разных подхода к загрузке модулей и, в частности, синтаксису загрузки модулей. Эти два стиля синтаксиса были известны как **CommonJS** (используется в Node) и **AMD**, или **асинхронное определение модуля** (используется в Require). К счастью, TypeScript всегда поддерживал и CommonJS, и AMD.

Теперь, когда был опубликован синтаксис модуля ECMAScript 6, TypeScript принял и реализовал его и автоматически сгенерирует правильный синтаксис для CommonJS или AMD на основе одного параметра компилятора.

В этой главе мы рассмотрим, что такое модули, взглянем на модульный синтаксис ECMAScript 6 и осветим различия между CommonJS и AMD. Затем мы подробнее рассмотрим, как Require использует AMD и как SystemJS позволяет использовать CommonJS в браузере. Потом мы обсудим модули CommonJS применительно к Node и создадим простое приложение Node, используя фреймворк Express. После создания приложения Node мы исследуем мир облачных провайдеров и пока-

жем, как создавать и развертывать приложение, для работы которого не требуется сервер.

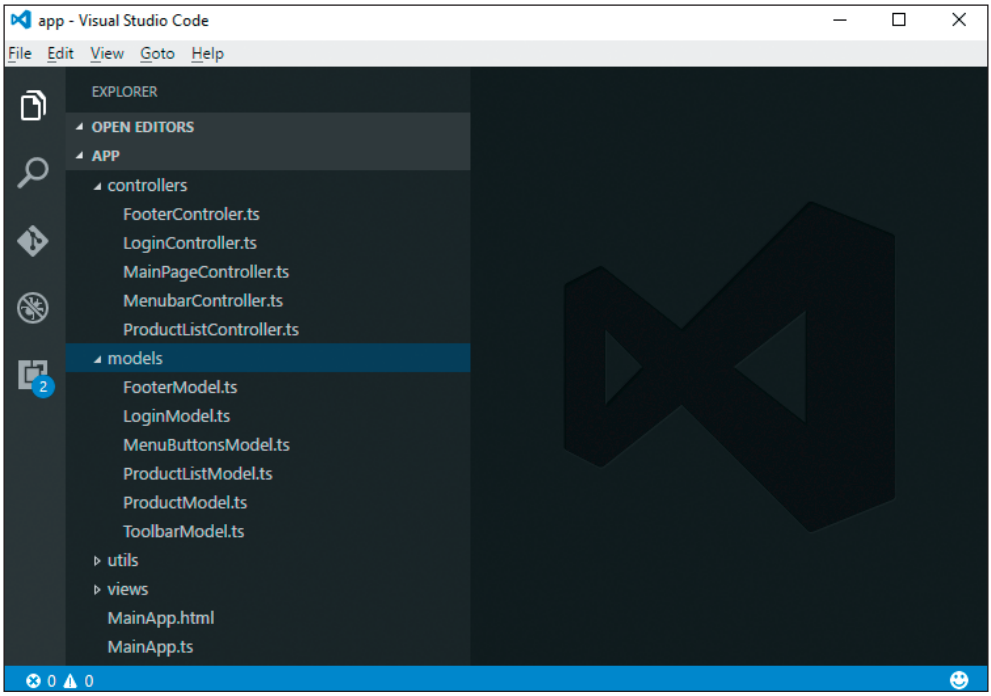
В этой главе мы рассмотрим следующие темы:

- что такое модуль;
- загрузка модуля AMD;
- загрузка модуля SystemJS;
- использование Express с Node;
- функции AWS Lambda.

Основы

Так что же такое модуль? По сути, **модуль** представляет собой отдельный файл TypeScript, который предоставляет классы, интерфейсы или функции для повторного использования в других частях проекта. Создание модулей помогает структурировать ваши файлы кода в логические группы. По мере того как ваше приложение становится все больше и больше, имеет смысл держать свои модели, представления, контроллеры, вспомогательные функции и т. д. в отдельных исходных файлах, чтобы их можно было легко найти.

Рассмотрим приведенное ниже дерево каталогов:



В этой структуре проекта у нас есть отдельный каталог для контроллеров, моделей, утилит и представлений. В каждом из этих каталогов у нас есть несколько файлов. Каждое имя файла является четким указанием того, что мы ожидаем от файла. Например, ожидается, что файл `FooterControler.ts` будет содержать класс контроллера, который обрабатывает нижний колонтитул нашего приложения. Эта структура делает наше программирование намного проще.

Проблема с таким количеством исходных файлов заключается в том, что для работы приложения наша HTML-страница должна обращаться к каждому файлу. Учитывая предыдущую структуру каталогов, она должна будет именовать каждый файл как исходный сценарий:

```
<html>
   <head>
      <script src="./Main.js"></script>
      <script src="./controllers/FooterController.js"></script>
      <script src="./controllers/LoginController.js"></script>
      <script src="./controllers/MainPageController.js"></script>
      <script src="./controllers/MenuBarController.js"></script>
      <script src="./controllers/ProductListController.js">
        </script>
      <!-- все другие файлы здесь ..-->
      <script src="./views/ToolbarView.js"></script>
   </head>
   <body>
   </body>
</html>
```

Включение каждого файла JavaScript в HTML-страницу занимает много времени и может привести к возникновению ошибок.

Чтобы преодолеть эту проблему, доступны два варианта: либо использовать процесс связывания, либо использовать загрузчик модулей. По сути, связывание означает, что мы запускаем шаг после компиляции, чтобы скопировать (или связать) все исходные файлы в один файл, поэтому нам нужно включить только один файл в нашу HTML-страницу. Хотя это и верное решение проблемы, оно означает, что HTML-страница должна загрузить весь связанный файл за один раз, прежде чем веб-страница будет готова к визуализации. Если это файл большого размера, это означает, что нашему браузеру нужно будет дождаться загрузки файла, что может повлиять на общее время загрузки страницы.

Загрузчики модулей, с другой стороны, позволяют браузеру загружать все файлы одновременно в отдельных потоках, что означает, что время загрузки нашей страницы значительно сокращается. Загрузчики модулей также дают возможность каждому из наших отдельных исходных файлов JavaScript определять, от каких файлов они зависят. Другими словами, если наша HTML-страница загружает файл

Main.js, а файл Main.js указывает, что ему нужен файл FooterController.js, а также файл MainPageController.js, загрузчик модулей обеспечит загрузку этих двух файлов перед выполнением логики в файле Main.js. Этот метод, по существу, позволяет нам определять дерево зависимостей для исходного файла.

Как только исходный файл был загружен загрузчиком модуля, любой файл, который зависит от этого файла, не будет нуждаться в браузере для перезагрузки файла с сайта. Это сводит к минимуму количество запросов к веб-серверу и ускоряет загрузку нашей страницы.

Экспорт модулей

Для написания и использования модулей нам нужны две вещи. Во-первых, модуль должен быть виден для внешнего мира, чтобы его можно было использовать. Это называется **экспортом** и использует ключевое слово TypeScript export. Это означает, что в конкретном исходном файле у вас могут быть функции и классы, которые считаются внутренними и не должны быть доступны для внешнего мира. Только компоненты, предназначенные для использования вне исходного файла, должны быть экспортированы. В качестве примера рассмотрим модуль, написанный в файле lib/Module1.ts:

```
export class Module1 {
   print() {
      print(`Module1.print()`);
   }
}

function print(functionName: string) {
   console.log(`print() called with ${functionName}`);
}
```

Здесь у нас есть класс Module1 и функция print в том же исходном файле. Однако класс Module1 добавил в определение класса ключевое слово export и поэтому будет доступен для использования внешним миром.

Функция print, однако, не использует ключевое слово export. Это означает, что она доступна только для использования в исходном файле Module1.ts и недоступна для использования внешним миром. Поэтому эта функция является закрытой с точки зрения области видимости. Класс Module1 очень прост и определяет функцию print. Внутри функции Module1.print выполняется вызов закрытой функции print, определенной в конце файла.

Таким образом, ключевое слово export предоставляет весь класс Module1 для использования внешним миром.

Импорт модулей

Чтобы использовать модуль, который был экспортирован, любой исходный файл, который нуждается в этом модуле, должен импортировать модуль, используя ключевое слово `import`. В предыдущем примере, если мы хотим использовать класс `Module1`, нам нужно импортировать его:

```
import {Module1} from './lib/Module1';

let mod1 = new Module1();
mod1.print();
```

Это файл `main.ts`, который находится в корне проекта. В первой строке этого файла используется оператор import для импорта определения класса `Module1` из файла `lib/Module1`. Обратите внимание на синтаксис этого оператора. За ключевым словом `import` следует имя в фигурных скобках `{Module1}`, а затем ключевое слово `from`, за которым идет имя файла самого модуля. Имя модуля `{Module1}` совпадает с именем экспортированного класса в файле `./lib/Module1`. Обратите внимание, что мы не указываем расширение `.ts` или `.js` при импорте модулей. Загрузчик модулей позаботится об отображении нашего оператора в правильное имя файла модуля на диске.

После того как модуль был импортирован, мы можем использовать определение класса `Module1`, как обычно. В последних двух строках предыдущего фрагмента кода мы просто создаем экземпляр класса `Module1` и вызываем функцию `print`. Вывод этого кода выглядит так:

```
print() called with Module1.print()
```

Поскольку мы работаем в среде Node по умолчанию, нам нужно будет вызвать наш скомпилированный файл `main.js`, запустив `node main` из командной строки.

Переименование модулей

При импорте модуля мы можем переименовать экспортированный модуль:

```
import {Module1 as m1} from './lib/Module1';

let m1mod1 = new m1();
mod1.print();
```

Мы импортировали тот же модуль из `'./lib/Module1'`, но использовали ключевое слово `as` при указании имени модуля, то есть `{Module1 as m1}`. Это означает, что теперь мы можем ссылаться на класс `Module1` (согласно нашему определению export) как просто `m1`. Последние две строки этого примера показывают, как

мы можем теперь создать класс (типа `Module1`), используя новое имя `m1`. Вывод этого примера точно такой же, как и в предыдущем случае:

```
print() called with Module1.print()
```

У нас также может быть несколько имен для экспортируемого модуля, но эти имена должны быть указаны в самом модуле. Рассмотрим приведенное ниже определение модуля:

```
export class Module1 {
   print() {
      print(`Module1.print()`);
   }
}

export {Module1 as NewModule};
```

В последней строке этого фрагмента кода класс `Module1` также был экспортирован с именем `NewModule`. Это позволяет потребителю использовать имя `Module1` или `NewModule` при импорте модуля:

```
import {NewModule} from './lib/Module1';
let nm = new NewModule();
nm.print();
```

Здесь мы импортируем класс `Module1`, используя имя `NewModule`, а затем используем имя класса `NewModule` для создания экземпляра класса `Module1`. Вывод этого кода такой же, как мы видели ранее:

```
print() called with Module1.print()
```

Экспорт по умолчанию

Когда файл модуля экспортирует только один элемент, мы можем пометить этот элемент как экспорт по умолчанию. Это достигается с помощью ключевого слова `default`. Рассмотрим файл модуля `lib/Module2.ts`:

```
export default class Module2Default {
   print() {
      console.log(`Module2Default.print()`);
   }
}

export class Module2NonDefault {
   print() {
      console.log(`Module2NonDefault.print()`);
   }
}
```

Здесь мы пометили класс `Module2Default` как экспорт по умолчанию для этого модуля. Обратите внимание, что у нас может быть только один экспорт по умолчанию для каждого модуля, но мы можем экспортировать другие элементы в файле, используя стандартный синтаксис экспорта. Это можно увидеть во втором экспорте класса `Module2NonDefault`.

Если у модуля есть экспорт по умолчанию, мы можем использовать более простой синтаксис для его импорта:

```
import Module2Default from './lib/Module2';

let m2default = new Module2Default();
m2default.print();
```

Мы удалили фигурные скобки `{...}` и импортируем экспорт по умолчанию как `Module2Default`. Имя, которое мы используем в операторе импорта, может быть любым, и его можно переименовать в операторе импорта:

```
import m2rn from './lib/Module2';

let m2renamed = new m2rn();
m2renamed.print();
```

Здесь мы импортируем экспорт по умолчанию из файла `'./lib/Module2'`, как было показано ранее, но переименовываем его в `m2rn`. Обратите внимание, что так же, как мы использовали ранее переименованные имена модулей, нам нужно будет ссылаться на модуль под новым именем, как видно из использования этого модуля, то есть `new m2rn()`.

Хотя в некоторых случаях это может служить цели, переименование модулей при импорте может затруднить чтение кода. Как правило, старайтесь сохранять имена модулей при импорте такими же, как и имена модулей, которые были экспортированы. Это помогает при чтении кода, точно зная, на какой модуль мы ссылаемся в исходном файле.

Экспорт переменных

Как и в случае с другими экспортируемыми элементами, мы также можем экспортировать переменные, которые были определены в модуле. Рассмотрим следующий экспорт в `lib/Module1.ts`:

```
var myVariable = "This is a variable.";
export { myVariable }
```

Здесь мы определяем переменную `myVariable` и устанавливаем значение в файле `Module1.ts`. Затем мы экспортируем саму переменную, заключив ее имя

в фигурные скобки, то есть `{myVariable}`. После этого мы можем импортировать и использовать эту переменную следующим образом:

```
import { myVariable } from './lib/Module1';
console.log(myVariable);
```

Хотя это может показаться немного странным и, на первый взгляд, нарушает принципы объектно-ориентированного программирования, данный метод используется многочисленными фреймворками для внедрения функциональности в существующие одиночные экземпляры. Мы рассмотрим эту технику позже в этой главе, когда будем обсуждать настройку маршрутов с помощью движка Express.

Наряду с переменными и функциями интерфейсы также можно экспортировать с помощью ключевого слова `exports`. В больших проектах, где используются конечные точки REST, существует ряд инструментов, которые могут автоматически генерировать интерфейсы, описывающие входы и выходы конечной точки REST. При использовании этих инструментов будет создано большое количество определений интерфейса, которые можно импортировать в наш код.

Типы импорта

Бывают моменты, когда нам нужно написать глобальный сценарий, который не использует модули. Это означает, что у нас нет возможности использовать ключевое слово import в нашем коде. Обратите внимание, что ключевое слово import делает для нас две вещи. Во-первых, оно импортирует файл объявлений для модуля, поэтому у нас есть доступ к сигнатурам типов экспортируемых переменных и функций через нашу среду разработки. Во-вторых, во время выполнения оно загружает файл JavaScript для использования. Когда мы не используем модули, нам нужно указать каждый из файлов JavaScript в нашем HTML-файле, как было показано ранее в этой главе.

К сожалению, при попытке написать код без модулей теряется способность среды разработки использовать файлы объявлений. Это означает, что мы не можем видеть, какие свойства, функции или переменные были экспортированы во время разработки. Однако если у нас есть доступ к файлам объявлений, мы можем импортировать только эти типы, используя так называемые типы импорта.

В качестве примера давайте создадим файл объявлений для модуля `Module3`. Он будет создан в файле `Module3.d.ts`:

```
export declare class Module3 {
   print(): void;
   add(): void;
   remove(): void;
```

```
    id: number;
    name: string;
}
```

Здесь мы объявили класс с именем `Module3`, у которого есть три функции: `print`, `add` и `remove` – и свойства `id` и `name`. Давайте теперь создадим функцию, которая использует этот класс в немодульном файле `import_types.ts`:

```
function printWithoutModule(mod3: any) {
    mod3.print();
}
```

Здесь у нас есть функция `printWithoutModule` с единственным аргументом `mod3`. Нам нужно указать тип этого класса как any, чтобы использовать его. К сожалению, использование типа any означает, что мы лишаемся безопасности всех типов.

Однако мы можем использовать ключевое слово import для импорта файла объявлений этого класса:

```
function printModule3(mod3: import("./Module3").Module3) {
    mod3.print();
}
```

Здесь мы использовали ключевое слово `import` для импорта нашего файла `Module3.d.ts` и корректного приведения типа `mod3` к определению `Module3`. Это означает, что наш редактор автоматически подберет правильные функции и свойства класса `Module3`.

Мы импортировали только те типы, которые были найдены в файле объявлений для использования в нашем коде.

Асинхронное определение модуля

Синтаксис экспорта и импорта модулей, который мы использовали до сих пор, использует так называемый синтаксис CommonJS и является механизмом загрузки модулей по умолчанию при использовании Node. Традиционно этот синтаксис не был доступен для использования в браузере, и поэтому стала популярной альтернатива CommonJS под названием асинхронное определение модуля, или AMD (Asynchronous module definition). Одна из наиболее распространенных библиотек для использования AMD – это RequireJS, или просто Require. В этом разделе мы будем повторно применять исходный код для модулей, которые мы создали в Node, и перекомпилировать их для использования с AMD. Затем мы покажем, как применять Require для загрузки этих модулей в браузере.

Компиляция

Чтобы скомпилировать наш код для использования синтаксиса модуля AMD, нам нужно изменить настройку модуля в файле tsconfig.json:

```
{
   "compilerOptions": {
      "module": "amd",
      "target": "es5",
      "noImplicitAny": false,
      "sourceMap": false
   },
   "exclude": [
      "node_modules"
   ]
}
```

Здесь мы указали параметр "module" как "amd" вместо "commonjs". Это изменение указывает компилятору TypeScript, что выходные данные шага компиляции должны генерировать код JavaScript, который использует синтаксис модуля AMD. Давайте посмотрим, что это означает с точки зрения наших файлов модулей.

Рассмотрим следующее определение модуля TypeScript, найденное в файле lib/Module3.ts:

```
export class Module3 {
   print() {
      console.log(`Module3.print()`);
   }
}
```

Здесь мы экспортируем класс Module3. Когда мы компилируем этот класс, используя опцию модуля CommonJS, TypeScript генерирует следующий JavaScript-файл:

```
"use strict";
var Module3 = /** @class */ (function () {
   function Module3() { }
   Module3.prototype.print = function () {
      console.log("Module3.print()");
   };
   return Module3;
}());
exports.Module3 = Module3;
```

В этом сгенерированном файле у нас есть стандартный шаблон замыкания, используемый для определения класса Module3. Однако обратите внимание на

последнюю строку файла. TypeScript сгенерировал строку `export.Module3 = Module3;`, которая присоединит класс `Module3` к переменной `export`. Это стандартный способ создания модулей при использовании JavaScript.

Если мы изменим наши параметры компиляции на `"amd"` вместо `"commonjs"`, TypeScript создаст следующий код для того же файла `lib/Module3.ts`:

```
define(["require", "exports"], function (require, exports) {
   "use strict";
   var Module3 = /** @class */ (function () {
      function Module3() {
      }
      Module3.prototype.print = function () {
         console.log("Module3.print()");
      };
      return Module3;
   }());
   exports.Module3 = Module3;
});
```

Присмотревшись к этому файлу, мы видим, что определение внутреннего класса для замыкания `Module3` и строка `exports.Module3 = Module3;` точно те же, что мы видели ранее. Однако все определение класса было заключено в функцию `define`. В этом разница между модулями CommonJS и AMD. AMD использует функцию `define`, которая принимает два параметра: массив строк и определение функции.

Массив строк, то есть `["require", "exports"]`, фактически является массивом зависимостей, который указывает, какие библиотеки необходимо загрузить перед попыткой загрузки этого модуля. Определение функции вызывается после загрузки зависимых библиотек. Кроме того, каждый из элементов, указанных в массиве зависимостей, становится параметрами, доступными в функции обратного вызова. Следовательно, `function (require, exports)` обеспечивает доступ к аргументам `require` и `export` внутри функции обратного вызова. Имея доступ к глобальной переменной `export`, мы можем теперь присоединить определение класса `Module3` к переменной `export`, которая была передана в качестве аргумента.

Мы никак не меняли наш TypeScript-файл `Module3.ts` для поддержки синтаксиса загрузки модулей CommonJs и AMD. Компилятор TypeScript автоматически позаботился об определениях модуля.

Установка модуля AMD

Теперь, когда наши модули компилируются в синтаксисе AMD, мы можем сосредоточить наше внимание на загрузке и использовании их в браузере. Для загрузки и использования модулей AMD в браузере мы будем использовать загрузчик

модулей Require. Require – это стандартный фреймворк JavaScript, и как таковой его можно установить через npm. После установки в нашем проекте нам также понадобятся соответствующие файлы объявлений. Мы можем установить Require, используя npm следующим образом:

```
npm install requirejs --save
```

Затем установите файлы объявлений:

```
install @types/requirejs --saveDev
```

После установки Require и файла объявлений мы можем теперь настроить Require для загрузки наших модулей AMD.

Настройка Require

Require использует глобальный файл конфигурации, обычно называемый `RequireConfig.js`, который служит точкой входа в наше браузерное приложение. Давайте продолжим и создадим файл TypeScript с именем `RequreConfig.ts`:

```
require.config( {
   });

require(['main'], (main: any) => {
   console.log(`inside main`);
});
```

Здесь мы начнем с вызова функции `require.config`, передавая объект конфигурации, который устанавливает значения по умолчанию. На этом этапе мы вызываем функцию `require.config` с пустым объектом. Мы более подробно рассмотрим доступные параметры конфигурации чуть позже в этой главе.

После вызова `require.config` мы вызываем функцию `require` с двумя аргументами. Функция `require` очень похожа на функцию `define`, которую мы видели ранее. Первый параметр `require` – это массив строк, в котором перечислены загружаемые файлы, а вторым параметром является функция обратного вызова. Как и с функцией `define`, каждая зависимость, указанная в первом массиве строк, будет доступна в функции обратного вызова в качестве параметра.

В нашем примере единственной перечисленной зависимостью является `main`, которая транслируется в наш файл `main.js`. Обратите внимание, что Require автоматически добавит расширение `.js` при попытке загрузить файлы JavaScript. После загрузки файла `main.js` она будет доступна в определении функции в качестве аргумента `main`.

Настройка браузера

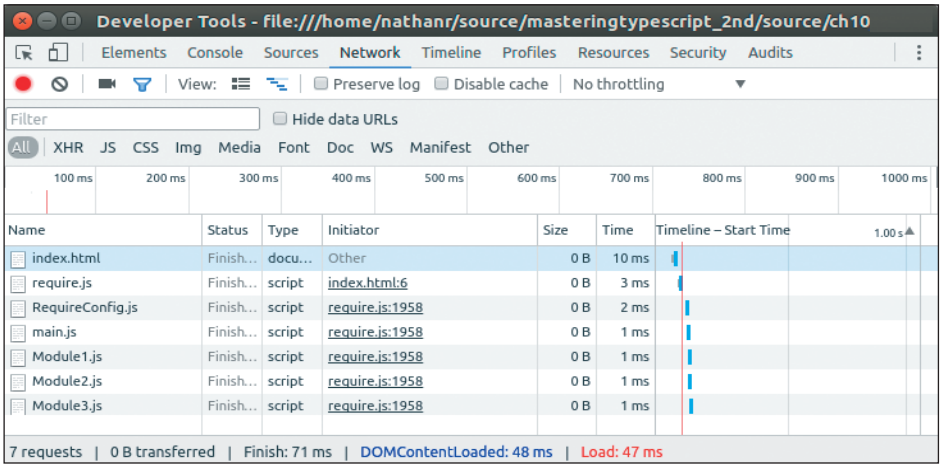
Таким образом, единственной оставшейся задачей является включение файла `RequireConfig.js` в наш HTML-код:

```
<html>
<head>
<script>
   type="text/javascript"
   src="./node_modules/requirejs/require.js"
   data-main="./RequireConfig"
</script>
</head>
<body>
</body>
</html>
```

Это очень простой HTML-файл, у которого есть только один тег `<script>` для загрузки файла `"./node_modules/requirejs/require.js"`. Это заставит браузер загрузить загрузчик модуля `require.js`. Однако обратите внимание, что у этого тега есть атрибут `data-main`. Этот атрибут используется Require для загрузки исходного файла конфигурации, в нашем случае это `RequireConfig.js`. И снова Require автоматически добавит расширение .js для файлов, поэтому этот атрибут просто указывается как `datamain = "./RequireConfig"`.

После загрузки файла `require.js` файл `RequireConfig.js` будет загружен и выполнен, и начнется процесс загрузки модуля.

Если мы используем вкладку **Network** (**Сеть**) в инструментах разработчика в браузере, то увидим, что Require загружает и анализирует каждый из файлов модуля и автоматически загружает их для использования:



Браузер начинает с загрузки файлов `index.html` и `require.js`, а затем мы видим, что он загружает наш файл `RequireConfig.js`. В нем мы указали, что нам нужно загрузить `main.js`, поэтому это следующий файл, который Require загрузит. Этот файл `main.js` затем использовал синтаксис модуля импорта для импорта файлов, `lib/Module1` и `lib/Module2`. После этого видно, что эти два файла загружаются следующими.

Интересно, что файл `lib/Module1` также импортирует файл `lib/Module3`, поэтому Require загружает и его тоже.

Как видно из диагностики сети, Require рекурсивно анализирует каждый наш модульный файл, начиная с `RequireConfig.js`, и динамически загружает все найденные модули. Таким образом, мы можем определять модули по своему усмотрению, просто используя синтаксис модулей `export` и `import`. Пока мы импортируем наши зависимости внутри модуля, загрузчик модулей будет автоматически загружать эти файлы от нашего имени.

Зависимости модуля AMD

При работе с модулями часто бывает так, что один модуль должен быть загружен раньше другого. Когда модуль B уже должен иметь загруженный модуль A, можно сказать, что модуль B имеет зависимость от модуля A. При построении стандартной HTML-страницы эту зависимость довольно легко получить. Все, что нам нужно сделать, – это убедиться, что тег `<script>` модуля A включен в веб-страницу над тегом `<script>` модуля B. К сожалению, это немного сложнее при использовании загрузки модуля AMD.

При загрузке модуля AMD каждый модуль загружается независимо и асинхронно. Это означает, что порядок, в котором мы указываем наши модули, недостаточен. В этом случае нам нужно уметь описывать зависимости между модулями, чтобы загрузчик модулей AMD мог координировать эти запросы.

Загрузчик модулей Require использует параметры в вызове `require.config` для определения зависимостей, а также других характеристик модуля. В качестве примера необходимости загружать файлы в очень конкретном порядке давайте настроим среду тестирования Jasmine с использованием загрузки модулей AMD.

Если вы помните в предыдущей главе, мы создали файл `SpecRunner.html` для модульного тестирования образцов Backbone. Этот файл загрузил три базовых файла:

```
<script type="text/javascript"
   src="./<path_to_jasmine>/jasmine.js" >
</script>
<script type="text/javascript"
   src="./<path_to_jasmine>/jasmine-html.js" />
</script>
```

```
<script type="text/javascript"
   src="./<path_to_jasmine>/boot.js" >
</script>
```

Фреймворк Jasmine содержит три файла компонентов, которые необходимо загрузить в правильном порядке: сначала `jasmine.js`, потом `jasmine-html.js` и, наконец, `boot.js`. Загрузка `boot.js` до `jasmine.js` приведет к ошибкам во время выполнения, поэтому `boot.js` зависит от `jasmine.js`. Давайте посмотрим на файл `RequireConfigSpecRunner.ts`, который показывает, как выглядит файл `require.config` для среды Jasmine:

```
require.config( {
   baseUrl: ".",
   paths: {
      'jasmine':
        './node_modules/jasmine-core/lib/jasmine-core/jasmine',
      'jasmine-html':
        './node_modules/jasmine-core/lib/jasmine-core/jasmine-html',
      'jasmine-boot':
        './node_modules/jasmine-core/lib/jasmine-core/boot'
   },
   shim : {
      'jasmine': {
        exports: 'window.jasmineRequire'
      },
      'jasmine-html': {
        deps: ['jasmine'],
        exports: 'window.jasmineRequire'
      },
      'jasmine-boot': {
        deps: ['jasmine-html'],
        exports: 'window.jasmineRequire'
      }
   }
});
```

Здесь мы включили два новых свойства в наш вызов `require.config`, а именно `paths` и `shim`. Мы обсудим свойство `shim` чуть позже, но пока сосредоточимся на свойстве `paths`. Свойство `paths` содержит запись свойства для каждого из наших файлов Jasmine. Здесь важно отметить, что это именованные записи и что имя записи должно использоваться в остальной части конфигурации Require. Если мы посмотрим на первую запись, которая называется 'jasmine' и указывает на '`./node_modules/jasmine-core/lib/jasmine-core/jasmine`', то имя этой записи будет '`jasmine`', и все ссылки на данный файл с этого момента должны использовать имя '`jasmine`'. Мы могли бы легко назвать ее '`jasminejs`' или '`jjs`', если имя записи используется последовательно во всей конфигурации Require.

Запись 'jasmine-boot' является прекрасным примером этой схемы именования, поскольку сам файл называется просто boot.js, а не jasmine-boot.js, но именованная запись – 'jasmineboot'.

Также обратите внимание, что Require добавит расширение файла .js к каждой из этих записей при загрузке файла с диска.

Следующий блок конфигурации – это свойство shim. Это свойство содержит запись для каждой из наших именованных библиотек. Запись shim для каждой из этих библиотек может содержать записи exports и/или deps. Запись exports используется для указания глобального пространства имен JavaScript, к которому будет подключена эта библиотека. В качестве простого примера того, что должно содержать свойство exports, рассмотрим следующие записи shim для jQuery, Underscore и Backbone (обратите внимание, что они не включены в наш текущий require.config, а показаны здесь в иллюстративных целях):

```
'jquery': {
  exports: '$'
},
'underscore': {
  exports: '_'
},
'backbone': {
  exports: 'Backbone'
}
```

Свойство exports библиотеки jQuery – это просто $. Это означает, что библиотека jQuery присоединяется к пространству имен $ с помощью Require, что позволяет нам использовать любую функцию jQuery, добавляя к ней префикс $, как в $('#elementId') . Аналогично, библиотека Underscore использует символ _ в качестве своего пространства имен, и он используется простым вызовом _.bind (...). В качестве последнего примера библиотека Backbone применяет пространство имен Backbone, которое используется, например, путем вызова new Backbone.Model. Поэтому каждая из этих библиотек определяет глобальное пространство имен в свойстве exports.

Наряду со свойством exports в нашей конфигурации shim каждый из наших модулей может также указывать запись deps, которая представляет собой массив строк. Запись deps используется для описания зависимостей модуля. Если мы начнем с нижней части записей shim, то увидим, что запись 'jasmine-boot' определяет запись 'jasmine-html' в качестве зависимости. Точно так же запись 'jasmine-html' определяет 'jasmine' как зависимость. Поэтому Require примет во внимание эти зависимости и загрузит наши модули по порядку.

Свойство deps является массивом строк, что означает, что одна запись может указывать несколько зависимостей.

Начальная загрузка Require

Как мы видели в нашей минимальной реализации ранее, процесс загрузки модуля начинается с первоначального вызова функции `require`. Предполагая, что у нас есть очень простой тест Jasmine в файле `test/SimpleTest.ts`, мы можем выполнить начальную загрузку нашей тестовой среды в нижней части файла `RequireConfigSpecRunner.js`:

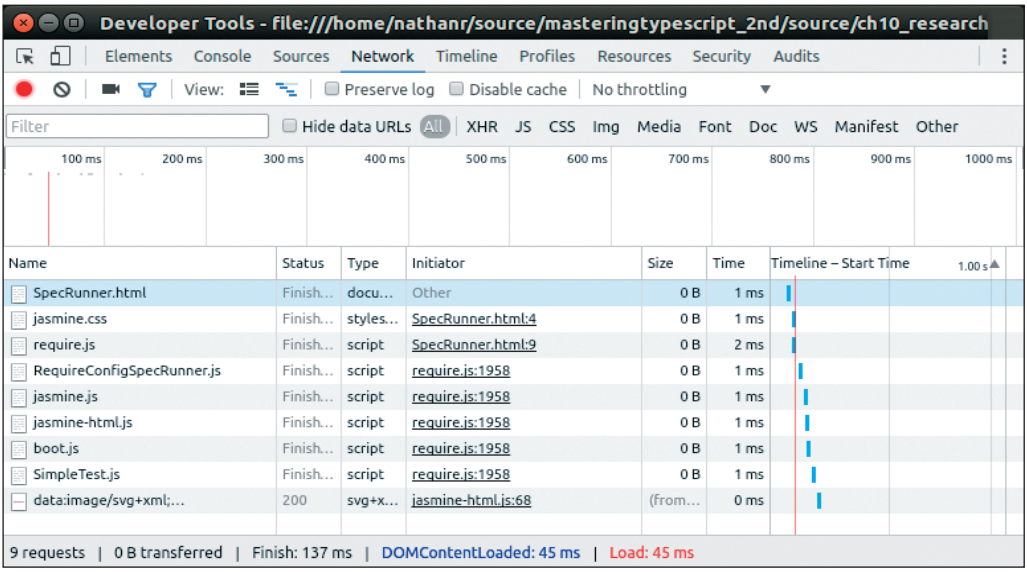
```
var specs = [
   'test/SimpleTest'
];

require(['jasmine-boot'], (jasmineBoot) => {
   require(specs, () => {
      (<any>window).onload();
   });
});
```

Эта настройка интересна по нескольким причинам. Во-первых, мы определили переменную `specs`, которая представляет собой простой строковый массив. Она содержит одну запись, а именно '`test/SimpleTest`', которая является ссылкой на наш набор тестов Jasmine. Теперь обратите внимание, где используется эта переменная. Она используется в вызове `require`, который вложен во внешний вызов `require`. Внешний вызов говорит `require`, чтобы он загружал модуль '`jasmine-boot`' перед выполнением функции обратного вызова. Поскольку запись shim модуля '`jasmine-boot`' задает путь зависимости, функция обратного вызова будет выполняться только после того, как все зависимости будут выполнены.

Как только внешняя функция обратного вызова выполнена, тело этой функции снова вызывает функцию `require`, но на этот раз с массивом, который перечисляет все файлы Jasmine в нашем наборе. Этот внутренний вызов Require уже будет загружать зависимые модули (то есть файлы Jasmine) до его выполнения. После загрузки всех модулей в массиве `specs` функция обратного вызова просто вызывает `window.onload()`, которая запускает прогон тестов Jasmine.

Также обратите внимание, что глобальная переменная `window` должна быть приведена к типу `<any>`, чтобы разрешить компиляцию TypeScript.

После проведенных манипуляций мы можем открыть наш браузер и запустить файл `SpecRunner.html` для выполнения всех тестов. Опять же, запустив наши сетевые инструменты для разработчиков, можно увидеть порядок загрузки каждого из наших модулей:



Этот порядок загрузки модуля соответствует тому, что определено в файле `require.config`, и использует свойство `deps` каждой из записей для определения порядка загрузки файлов. Сначала загружается файл `SpecRunner.html` вместе с `jasmine.css` и, наконец, сам файл `require.js`. Мы указали, что Require должен загрузить файл `RequireConfigSpecRunner` с помощью атрибута `data-main` в нашем HTML-файле. Поэтому Require загружает этот файл и начинает интерпретировать записи в нем. Обратите внимание, что сначала он загружает `jasmine.js`, который не имеет зависимостей, а затем загружает `jasmine-html`, который имеет зависимость от `Jasmine`. После этого он загружает `boot.js`. После того как все зависимости внутри файла `require.config` загружены, он запустит внутренний код require, который загружает `SimpleTest.js`.

Исправление ошибок конфигурации

Довольно часто при разработке приложений AMD с использованием Require мы можем встречать неожиданное поведение, получать странные сообщения об ошибках или просто пустые страницы. Эти странные результаты обычно вызваны конфигурацией Require либо в свойствах `paths`, `shim` или `deps`. Поначалу исправление этих ошибок может быть довольно неприятным, но, как правило, они вызваны одной из двух причин – неправильными зависимостями или ошибками, когда файл не найден.

Чтобы исправить эти ошибки, нам нужно будет открыть инструменты отладки в используемом нами браузере, что в большинстве браузеров делается простым нажатием клавиши *F12*.

Неправильные зависимости

Некоторые ошибки AMD вызваны неправильными зависимостями в `require.config`. Эти ошибки можно найти, проверив вывод консоли в браузере. Ошибки зависимости приводят к ошибкам браузера, подобным этим:

```
ReferenceError: jasmineRequire is not defined
ReferenceError: Backbone is not defined
```

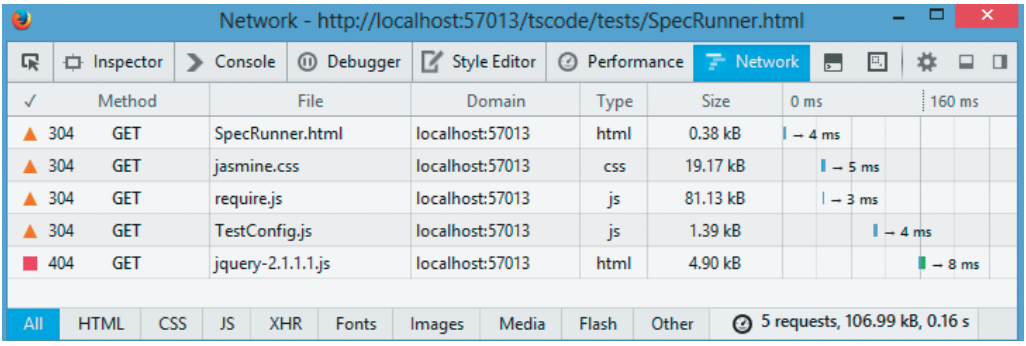
Этот тип ошибки может означать, что загрузчик AMD загрузил Backbone, например, перед загрузкой Underscore. Таким образом, всякий раз, когда Backbone пытается использовать функцию Underscore, мы получаем ошибку `not defined`, как показано в предыдущем выводе. Исправить такую ошибку можно, обновив свойство библиотеки `deps`, которое вызывает ошибку. Убедитесь, что все обязательные библиотеки были названы в свойстве `deps`, и ошибки должны исчезнуть. Если этого не происходит, то речь идет о следующем типе ошибки AMD, `file-not-found`.

Ошибки 404

Файл не найден, или ошибки 404 обычно указываются в выводе консоли:

```
Error: Script error for: jquery
http://requirejs.org/docs/errors.html#scripterror
Error: Load timeout for modules: jasmine-boot
http://requires.org/docs/errors.html#timeout
```

Чтобы узнать, какой файл вызывает предыдущую ошибку, перейдите на вкладку **Сеть** в инструментах отладчика и обновите страницу. Найдите ошибки 404 (файл не найден), как показано на скриншоте ниже:



На этом скриншоте видно, что вызов jquery.js генерирует ошибку 404, поскольку наш файл на самом деле называется `/Scripts/jquery-2.1.1.js`. Ошибки такого рода можно исправить, добавив запись в параметр paths в `require.config`, чтобы любой вызов `jquery.js` заменялся вызовом `jquery-2.1.1.js`.

У Require есть неплохой набор документации для распространенных ошибок AMD (`http://requirejs.org/docs/errors.html`), а также расширенные возможности использования API, включая циркулярные ссылки (`http://requirejs.org/docs/api.html#circular`), поэтому обязательно посетите сайт для получения дополнительной информации о возможных ошибках AMD.

Загрузка модулей с помощью SystemJS

SystemJS – это загрузчик модулей, который понимает формат модулей CommonJS, AMD и даже новый формат модулей ES6. Он работает как в Node, так и в браузере и поэтому позиционирует себя как универсальный загрузчик модулей. До появления SystemJS решения на основе Node использовали формат CommonJS, а решения на основе браузера применяли формат AMD. Теперь мы можем использовать формат CommonJS в браузере и даже смешивать и сочетать синтаксис модулей. В этом разделе мы рассмотрим, как настроить SystemJS для загрузки модулей CommonJS в браузере.

Установка SystemJS

SystemJS можно установить с помощью npm:

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

Соответствующие файлы объявлений могут быть установлены с помощью `@types`:

```
npm install @types/systemjs --saveDev
```

Конфигурация браузера

Чтобы использовать SystemJS в нашем браузере, нам нужно будет включить исходный файл `system.js`, а затем запустить сценарий конфигурации SystemJS, аналогичный нашему файлу `RequireConfig.js`. Наша HTML-страница выглядит так:

```
<html>
<head>
</head>
<body>
   <script src=
      "./node_modules/systemjs/dist/system.js">
   </script>
   <script src="./SystemConfig.js"></script>
</body>
</html>
```

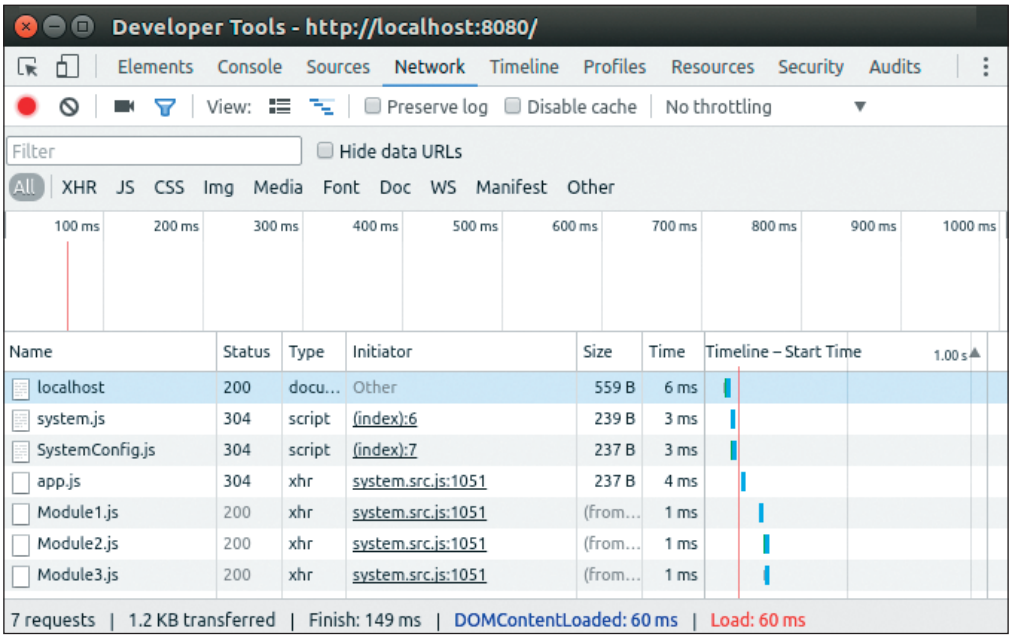
Здесь мы включили два файла сценария. Один для самого фреймворка `system.js`, а другой для файла `SystemConfig.js`. Этот файл генерируется из файла `SystemConfig.ts`:

```
SystemJS.config({
  packages : {
    'lib' : {defaultExtension: 'js'}
  }
});
SystemJS.import('app.js');
```

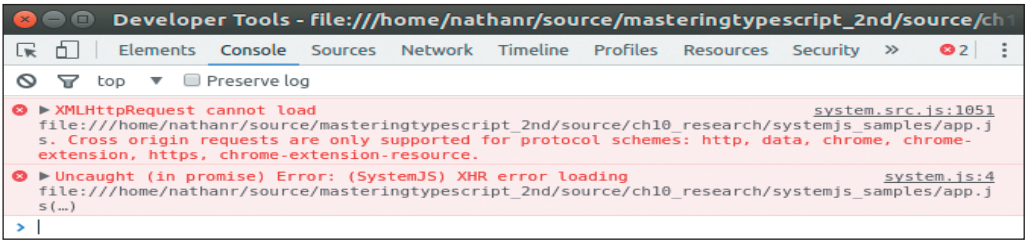
Наш файл конфигурации SystemJS начинается с вызова функции `SystemJS.config` и включает в себя объект конфигурации. В этом объекте мы указали только одно свойство, `packages`. Это свойство указывает свойство `lib` и, в рамках него, свойство `defaultExtension: 'js'`. SystemJS использует свойство `packages`, чтобы указать параметры для каждого из наших исходных каталогов или пакетов. Поэтому свойство `lib` относится ко всем файлам, содержащимся в каталоге `./lib`. Свойство `defaultExtension` сообщает SystemJS, что все модули в подкаталоге `./lib`. по умолчанию имеют расширение `.js`.

Это означает, что когда SystemJS сталкивается с импортом модуля, как, например, `import {Module1}` из `'./lib/module1'`, он добавляет расширение по умолчанию `'.js'` к имени файла модуля и, следовательно, загружает файл с именем `'./Lib/module1.js'`.

Вторая часть нашего файла конфигурации SystemJS – это вызов функции `SystemJS.import`, в которой файл `app.js` указывается в качестве отправной точки для нашего приложения. Как только SystemJS загрузит файл `app.js`, он начнет анализировать наш код для всех других импортированных модулей, а затем загружать их для использования. Если мы просмотрим вкладку **Сеть** в панели **инструментов веб-разработчика**, то увидим загрузку следующих файлов:



Обратите внимание, что как в случае с модулями AMD, так и в случае с модулями SystemJS браузер Chrome предполагает, что они обслуживаются с работающего веб-сервера. Если вы попытаетесь загрузить страницу с диска, то есть нажать комбинацию клавиш *Ctrl+O* в Chrome и перейти к файлу на диске, то получите ряд ошибок, как показано на приведенном ниже скриншоте:



Другие браузеры, такие как Firefox, не показывают эти ошибки. Однако Chrome предоставляет параметр командной строки, который позволяет такое поведение, `--allow-file-access-from-files`.

В качестве альтернативы, чтобы загрузить страницу, которая использует SystemJS, просто запустите `http-server` из корневого каталога вашего проекта, а затем перейдите к URL-адресу, показанному на консоли. Когда `http-server` запустится, он покажет IP-адрес и порт, на который вы должны указать свой браузер:

```
Starting up http-server, serving ./
```

```
Available on:
  http://127.0.0.1:8080
  http://192.168.1.101:8080
```

Зависимости модулей

До сих пор мы показали, как настроить SystemJS для загрузки модулей в качестве зависимостей, когда они импортируются с использованием операторов `import` в нашем коде. Давайте завершим это обсуждение SystemJS, показав, как обращаться с зависимостями модулей, как мы это делали в случае с AMD. Как и в случае с AMD, мы создадим фреймворк для модульного тестирования с Jasmine и SystemJS. Помните, что у Jasmine есть определенный порядок загрузки модулей. Это означает, что основной файл `jasmine.js` должен быть загружен до `jasmine-html.js`, а затем, когда это будет сделано, мы можем загрузить модуль `boot.js` и запустить свои тесты.

Предположим, что у нас есть два очень простых набора тестов в каталоге `test` с именами `SimpleTest.ts` и `SimpleTest2.ts`. Эти два файла просто выполняют проверку работоспособности.

`SimpleTest.ts` выглядит так:

```
describe('SimpleTest.ts: sanity test', () => {
  it('should pass', () => {
    expect(true).toBeTruthy();
  });
});
```

А `SimpleTest2.ts` так:

```
describe('SimpleTest2.ts: sanity test 2', () => {
  it('should pass', () => {
    expect(true).toBeTruthy();
  });
});
```

Для запуска этих тестов нам также понадобится файл `SpecRunner.html`:

```
<html>
<head>
  <link rel="stylesheet"
    type="text/css" href="./node_modules/jasmine-core/lib
      /jasmine-core/jasmine.css" />
</head>
<body>
  <script src="./node_modules/system.js/dist/system.js">
  </script>
```

```
    <script src="./SystemConfigSpecRunner.js"></script>
</body>
</html>
```

Здесь у нас есть стандартный HTML-файл, который загружает файл `jasmine.css` и сам `system.js`.

Обратите внимание, что мы затем загружаем файл `SystemConfigSpecRunner.js`, который содержит нашу конфигурацию. Этот файл выглядит так:

```
SystemJS.config({
  baseUrl: '.',
  packages: {
    'lib': {defaultExtension: 'js'},
    'test': { defaultExtension: 'js'}
  },
  paths: {
    'jasmine': './node_modules/jasmine-core/lib/
      jasmine-core/jasmine.js',
    'jasmine-html': './node_modules/jasmine-core/lib/
      jasmine-core/jasmine-html.js',
    'jasmine-boot': './node_modules/jasmine-core/lib/
      jasmine-core/boot.js'
  },
  meta: {
    'jasmine-boot': {
      deps: ['jasmine-html'],
      exports: 'window.jasmineRequire'
    },
    'jasmine-html': {
      deps: ['jasmine'],
      exports: 'window.jasmineRequire'
    },
    'jasmine': {
      exports: 'window.jasmineRequire'
    }
  }
});

SystemJS.import('jasmine-boot').then( () => {
  Promise.all([
    SystemJS.import('test/SimpleTest'),
    SystemJS.import('test/SimpleTest2')
  ])
  .then(() => {
    (<any>window).onload();
  })
```

```
    .catch(console.error.bind(console));
});
```

Этот файл конфигурации состоит из двух частей. Сначала мы вызываем функцию `SystemJS.config` с помощью блока конфигурации, а затем, в нижней части файла, вызываем функцию `SystemJS.import` для загрузки нашей тестовой среды Jasmine. Давайте для начала сосредоточимся на блоке конфигурации.

Наши блоки конфигурации начинаются с указания свойства baseUrl как '.'. Это говорит SystemJS, что все запросы модуля относятся к текущему каталогу. Следующее свойство – это `packages`, и, как мы видели ранее в случае с SystemJS, оно устанавливает расширение по умолчанию `js` для подкаталогов '`lib`' и '`test`'.

Третье свойство в нашем блоке конфигурации – это свойство `paths`. Свойство это очень похоже на свойство paths в AMD, с одним заметным исключением – включением расширения файла `.js` в каждое из свойств пути. Как мы видели в версии Require, эти пути являются **именованными свойствами**, и поэтому имя, указанное в свойстве `paths` (например, '`jasmine`'), должно использоваться последовательно на протяжении всего блока конфигурации.

Следующее свойство, которое нам нужно, – это свойство `meta`. Формат и использование свойства `meta` в точности совпадают со свойством `shim`, используемым в Require, и делает оно то же самое. Свойство `meta` – это то, где установлены наши зависимости для каждой из библиотек Jasmine. Как и в Require, мы указываем свойство `deps` (для зависимостей) и свойство `exports` (для нашего глобального пространства имен).

Начальная загрузка Jasmine

Теперь давайте подробнее рассмотрим вызов `SystemJS.import` в нижней части файла `SystemConfigSpecRunner.ts`:

```
SystemJS.import('jasmine-boot').then( () => {
   Promise.all([
      SystemJS.import('test/SimpleTest'),
      SystemJS.import('test/SimpleTest2')
   ])
   .then(() => {
      (<any>window).onload();
   })
   .catch(console.error.bind(console));
});
```

Здесь мы выполняем начальную загрузку среды Jasmine, загружая модуль Jasmine с именем '`jasmine-boot`'. Как и в случае с Require, SystemJS найдет и загрузит все зависимости, которые были указаны в нашем дереве зависимостей для

'jasmine-boot', который в данном случае включает в себя как 'jasmine-html', так и 'jasmine'. Затем мы присоединяем функцию then, которую нужно выполнить после загрузки Jasmine. В рамках этой функции мы загружаем наши два набора тестов с помощью вызова Promise.all. Этот метод похож на тот, который мы использовали в Require, где мы разделили загрузку тестовых спецификаций за пределами нашего блока SystemJs config, чтобы было проще добавлять несколько тестов без значительных изменений конфигурации SystemJS. Функция Promise.all загружает все файлы спецификаций и снова использует свободный синтаксис для присоединения функции then, которая будет выполнена, когда все файлы будут загружены. Функция просто вызывает window.onload() и, как мы видели в Require, заставит Jasmine выполнить все тесты. Последний вызов – catch, где мы записываем ошибки в консоль.

Наша конфигурация SystemJS завершена. После этого мы можем загрузить файл SpecRunner.html для выполнения наших тестов Jasmine. Теперь мы знаем, как определять и использовать зависимости модулей с помощью SystemJS.

Использование Express с Node

В этом разделе мы продолжим исследование модулей, показывая, как создать простое приложение для веб-сервера на платформе Node. Для этого мы воспользуемся веб-фреймворком ExpressJS (или просто Express) для Node. Express предоставляет библиотеку многократно используемых модулей Node для обработки базовой функциональности, необходимой для создания веб-сервера. Она включает в себя маршрутизацию, шаблонизатор для создания веб-страниц, библиотеки для обработки сессий и файлов cookie, аутентификацию и сообщения об ошибках (например, ошибки 404) среди прочего. Express предоставляет богатый набор модулей и API-интерфейсов, которые включают в себя все, что вам потребуется при создании приложения для производственного веб-сервера.

Установка Express

Чтобы создать приложение Express, нам нужно будет установить Express в нашей среде разработки, а также включить различные файлы объявлений, необходимые для компиляции TypeScript. Express можно установить с помощью npm:

```
npm init
npm install express --save
```

После установки Express нам понадобятся соответствующие файлы объявлений:

```
npm install @types/express --saveDev
```

Express использует ряд других библиотек npm, файлы объявлений которых не включены в основной файл объявлений express.d.ts. Чтобы разрешить компиляцию TypeScript, нам нужно будет установить ряд других файлов объявлений:

```
npm install @ types / express-serve-static-core --saveDev
npm install @ types / serve-static --saveDev
npm install @ types / mime --saveDev
npm install @ types / node -saveDev
```

Теперь мы можем написать простейшее веб-приложение для Express в файле simple_app.ts:

```
import express from 'express';

let app = express();

app.get('/', (req: express.Request, res: express.Response) => {
   res.send('Hello World');
});

app.listen(3000, () => {
   console.log('listening on port 3000');
});
```

Мы начинаем с импорта модуля Express, from 'express' и присоединения его к пространству имен express. Обратите внимание, что мы используем экспорт по умолчанию с именем express из библиотеки express.

Затем мы создаем локальную переменную app и присваиваем ей новый экземпляр express().

Модуль Express, который мы импортировали, по умолчанию имеет функцию-конструктор, которую мы используем для создания нашего приложения Express.

Затем мы вызываем функцию get для экземпляра app. Она настроит так называемый обработчик маршрутов в Express. Первым аргументом является строка '/', которая сообщает Express, что любой HTTP-запрос к '/' должен обрабатываться нашей функцией-обработчиком, которая является вторым аргументом функции get. В этой функции мы просто вызываем функцию res.send для отправки строки 'Hello World' в HTTP-запрос. Обратите внимание, что мы строго типизируем параметр req как тип express.Request и параметр res как тип express.Response.

Express позволяет настроить несколько обработчиков маршрутов, так что '/login' может обрабатываться определенной функцией-обработчиком, а '/users' – другой. Если иные обработчики не указаны, Express направит запрос к ближайшему подходящему обработчику. Это означает, что если обработчик для '/login' определен, он будет обрабатывать все запросы, которые начинаются с '/login'. В на-

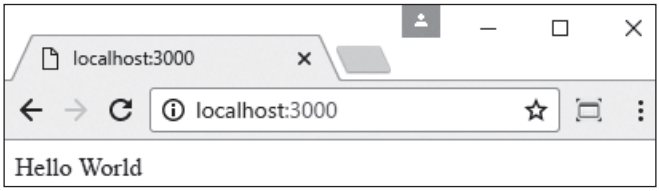
шем приложении мы указали обработчик только для '/', поэтому все запросы будут перенаправляться на этот обработчик.

В последней части нашего приложения мы вызываем функцию `listen` для экземпляра `app` и, по сути, устанавливаем цикл прослушивания. Первый аргумент – это номер порта для прослушивания, а второй – функция, которая вызывается при первоначальном запуске приложения. Здесь мы просто записываем сообщение в консоль.

Мы можем скомпилировать и затем запустить приложение Node Express, набрав:

```
tsc
node simple_app.js
```

Наше приложение запустится на порту `3000` и будет ожидать HTTP-запросов. Запуск браузера и указание его на `http://locahost:3000` вызовет обработчик запросов, который выведет в браузере фразу **Hello World**:



Использование модулей с Express

Если бы мы написали все обработчики для нашего приложения в одном файле, используя функцию `app.get` для каждого маршрута приложения, это очень быстро стало бы головной болью при обслуживании. Что нам действительно нужно сделать, это создать отдельный модуль для каждого обработчика запросов, а затем обращаться к ним из нашего основного приложения. К счастью, сделать это очень просто, используя стандартный синтаксис модуля.

В качестве примера давайте создадим функцию-обработчик в новом файле модуля. Назовем этот файл `SimpleModuleHandler.ts`:

```
export function processRequest(
   req : express.Request, res: express.Response) {
      console.log('SimpleModuleHandler.processRequest');
      res.send('Hello World');
   };
```

Здесь мы экспортируем функцию `processRequest`, которая является функцией-обработчиком запроса. Как у таковой у нее есть два параметра `req` и `res`, которые содержат объекты HTTP-запроса и ответа. Эта новая функция-обработчик

просто записывает сообщение в консоль и затем вызывает функцию `res.send` для записи строки в поток ответов, как мы делали ранее.

Наш файл `simple_app.ts` может быть изменен для использования этого модуля:

```
import express from 'express';
import { processRequest } from './SimpleModuleHandler';

let app = express();

app.get('/', processRequest );

app.listen(3000, () => {
   console.log(`listening on port 3000`);
});
```

Мы внесли два изменения в наше приложение Express. Во-первых, мы импортировали функцию processRequest из файла модуля с именем '`./SimpleModuleHandler`'.

Во-вторых, мы изменили вызов функции `app.get`. Вызов функции `app.get` теперь обращается к функции `processRequest` из импортированного модуля. Это означает, что когда наше приложение получит HTTP-запрос, он будет обработан функцией `processRequest` из модуля `SimpleModuleHandler`. Запуск этого приложения теперь будет записывать сообщение в консоль всякий раз при обработке запроса, как показано в следующем выводе консоли:

```
> node simple_module_app.js
listening on port 3000
SimpleModuleHandler.processRequest
```

Маршрутизация

Пока мы узнали, что можем зарегистрировать обработчик запросов для определенного HTTP-запроса. Однако в более сложных приложениях мы хотим иметь возможность регистрировать разные обработчики запросов в разных модулях, чтобы лучше структурировать свой код.

Express предоставляет так называемый обработчик маршрутов для этой цели. Мы можем создать множество разных обработчиков запросов и зарегистрировать их в глобальном экземпляре обработчика маршрутов Express.

Express предоставляет глобальный объект `Router` для обработки регистрации новых обработчиков маршрутов, а также для управления маршрутизацией приложений в целом. Давайте создадим два новых модуля `Login.ts` и `Index.ts` в каталоге routes. Мы будем использовать модуль `Index.ts` для обработки запросов к '`/`' и модуль `Login.ts` для обработки запросов к '`/login`'. Эта структура помогает нам разделить функциональность приложения на отдельные модули

и управлять нашим кодом. В производственном приложении у нас может быть большое количество различных маршрутов, каждый из которых написан в своих собственных отдельных модулях, и каждый из них обрабатывает запросы GET, POST или PUT.

Наш файл `Index.ts` в каталоге маршрутов будет выглядеть так:

```
import express from 'express';
var router = express.Router();

router.get('/', (req : express.Request, res: express.Response) => {
   res.send(`Index module processed ${req.url}`);
});

export { router } ;
```

Вначале идет импорт модуля `express`, как уже было ранее. Затем мы вызываем функцию `Router` для модуля express и присваиваем ее локальной переменной `router`. Функция `Router` действует как своего рода экземпляр-одиночка, что означает, что вызов `express.Router` возвращает один и тот же экземпляр маршрутизатора независимо от того, откуда он был вызван. Таким образом, мы можем присоединить новые маршруты к одному и тому же глобальному обработчику маршрутизатора Express и указать путь (`'/'`) и функцию обработчика маршрутов (`(req, res) => {}`) для каждого маршрута. В предыдущем примере наша функция обработчика маршрутов просто записывает сообщение в браузер.

Обратите внимание на последнюю строку этого модуля. Мы экспортируем переменную `router`, используя синтаксис экспорта переменных (`export{router}`). Помните, что эта переменная была установлена с помощью функции `express.Router()` в начале модуля, а затем использовалась для присоединения нового обработчика маршрутов при вызове `router.get`. Поскольку мы изменили этот глобальный экземпляр `router`, нам нужно реэкспортировать его, чтобы его могло использовать наше приложение. Это означает, что мы, по сути, присоединяем новый обработчик маршрутов к экземпляру-одиночке маршрутизатора Express.

Давайте теперь посмотрим на модуль `Login.ts`, который практически идентичен:

```
import * as express from 'express';
var router = express.Router();

router.get('/login', (req: express.Request,
   res: express.Response) => {
      res.send(`Login module processed ${req.url}`);
});

export { router };
```

Здесь модуль `Login.ts` также изменяет глобальный экземпляр `express.Router` и на этот раз подключает обработчик маршрутов для пути `'/login'`. Опять же, этот обработчик просто записывает сообщение в браузер. Последняя строка в этом модуле снова экспортирует переменную `router` через оператор `export{router}`. Затем Express дает нам возможность связать несколько обработчиков маршрутов с одним и тем же экземпляром `express.Router` путем импорта и повторного экспорта той же переменной `router`.

Давайте теперь обновим наше приложение, чтобы использовать эти два обработчика маршрутов:

```
import express from 'express';
let app = express();

import * as Index from './routes/Index';
import * as Login from './routes/Login';

app.use('/', Index.router);
app.use('/', Login.router);
app.listen(3000, () => {
   console.log(`listening on port 3000`);
});
```

Наше приложение теперь просто импортирует модули `Index` и `Login` из соответствующих файлов, а затем вызывает функцию `app.use` для регистрации наших обработчиков маршрутов. Обратите внимание, что мы обращаемся к экспортированной локальной переменной `router` из каждого модуля, как видно из вызова `Index.router` и `Login.router`. Поэтому эти две строки регистрируют наши обработчики маршрутов для нашего приложения.

При наличии этих модулей маршрутизации любой запрос веб-браузера к `'/'` будет обрабатываться модулем `Index.ts`, а запрос к `'/login'` будет обрабатываться модулем `Login.ts`. Таким образом, мы приступаем к организации своего кода в логические модули, каждый из которых отвечает за определенную область функциональности приложения.

Шаблонизаторы

Каждый из наших обработчиков маршрутов в настоящее время записывает в браузер очень простые сообщения. Однако в реальных приложениях нам нужно будет визуализировать HTML-страницы полностью. У этих страниц будет стандартная HTML-структура, они будут использовать таблицы стилей CSS и, в зависимости от креативности команды проектирования, с легкостью могут стать очень сложными. Для поддержки сложности генерации HTML-страниц большинство фреймворков предоставляет шаблонизатор.

Express также предоставляет фреймворк, который может использовать несколько различных шаблонизаторов, включая шаблоны Pug, Mustache, Jade, Dust и **Embedded JavaScript** (**EJS**) среди прочих. Внедрить шаблонизатор в наше приложение Express так же просто, как установить выбранный шаблонизатор и настроить Express для его использования.

В этом приложении мы будем использовать движок Handlebars. Handlebars применяет стандартные фрагменты HTML-кода и вводит переменные в HTML-шаблоны, используя простой синтаксис с двойными фигурными скобками `{{` и `}}`. Такие шаблонизаторы, как Pug или Jade, используют собственные пользовательские форматы для обозначения HTML-элементов, представляющих собой сочетание ключевых HTML-слов, имен классов и подстановки переменных. Для быстрого сравнения рассмотрим шаблон Handlebars:

```
<!DOCTYPE html>
<html>
   <head>
      <title>{{title}}</title>
      <link rel='stylesheet' href='/stylesheets/style.css' />
   </head>
   <body>
      {{{body}}}
   </body>
</html>
```

Этот шаблон очень похож на стандартный HTML-код с несколькими заменяемыми элементами, такими как `{{title}}` и `{{{body}}}`. Аналогичный шаблон Jade будет выглядеть так:

```
doctype
html
   head title #{title}
   link(rel='stylesheet', href='/stylesheets/style.css'
      body
```

Хотя шаблон Jade экономит нам много печатания, это означает, что нам нужно будет изучить и понять различные ключевые слова и тонкий синтаксис, используемые в Jade, для визуализации корректного HTML-кода. Обратите внимание, что здесь нет распознаваемых HTML-элементов, которые были заменены пользовательским синтаксисом Jade. Для простоты, а также чтобы избежать изучения совершенно нового синтаксиса HTML-шаблонов, мы будем использовать Handlebars в качестве нашего шаблонизатора, так как он использует распознаваемый HTML-синтаксис, перемежающийся с переменными подстановки.

Использование Handlebars

Handlebars можно установить через `npm`:

```
npm install hbs --save
```

После установки Handlebars нам нужно всего лишь добавить три строки в исходный файл нашего приложения (`app.ts`):

```
import express from 'express';
let app = express();

import * as Index from './routes/Index';
import * as Login from './routes/Login';

import * as path from 'path';
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'hbs');

app.use('/', Index.router);
app.use('/', Login.router);
```

Здесь мы добавили оператор импорта для модуля `path`. Модуль `path` позволяет использовать несколько удобных функций при работе с путевыми именами каталогов. Одной из переменных, предоставляемых модулем `path`, является переменная `__dirname`, которая содержит полный путь к текущему каталогу. Мы используем эту переменную при вызове функции `path.join`, которая будет возвращать полный путь к каталогу локальных представлений. Затем мы устанавливаем глобальный параметр Express `'views'` для этого каталога. Handlebars будет по умолчанию использовать этот глобальный параметр, чтобы найти путь, где хранятся файлы шаблона.

Наше последнее изменение в файле `app.ts` – это вызов app.set с аргументом `'view engine'` и значением `'hbs'`. Этот вызов функции указывает Express, что он должен использовать Handlebars в качестве шаблонизатора. Это все изменения, которые мы должны внести в наше приложение Express.

Теперь, когда мы зарегистрировали библиотеку шаблонов, мы можем обновить наш `routes/Index.ts`, чтобы использовать шаблон Handlebars:

```
import express from 'express';
var router = express.Router();

router.get('/', (req: express.Request, res: express.Response) => {
   res.render('index',
      {
         title: 'Express'
      }
   )
});

export { router };
```

Здесь мы обновили функцию обработчика маршрутов, чтобы вызывать `res.render` вместо `res.send`, как было использовано ранее. Функция `res.render` принимает имя шаблона в качестве первого параметра, а затем применяет простой Java-объект POJO в качестве входных данных для шаблонизатора.

Если мы запустим наше веб-приложение на этом этапе, появится ошибка, указывающая на то, что Handlebars не может найти представление с именем "`index`":

```
Error: Failed to lookup view "index" in views directory
"/express_samples/views" at EventEmitter.render
(//express_samples/node_modules/express/lib/application.js:579:17)
at ServerResponse.render
(//express_samples/node_modules/express/lib/response.js:960:7)
at //express_samples/routes/Index.js:7:9
at Layer.handle [as handle_request]
```

Теперь нам нужно создать шаблон представления `index`. Этот шаблон должен находиться в подкаталоге views и поэтому будет называться `views/index.hbs`. Handlebars использует расширение `.hbs` для указания файлов шаблонов Handlebars. Этот файл так же прост, как показано ниже:

```
<h1>{{title}}</h1>
<p>Welcome to {{title}}</p>
```

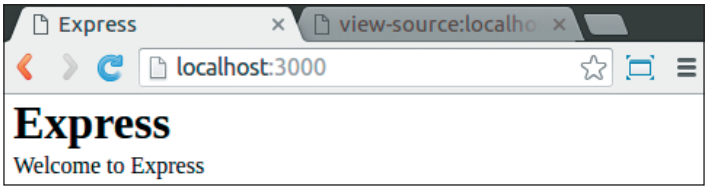
Наш файл шаблона `index.hbs` содержит элементы `<h1>` и `<p>`. Оба этих элемента используют аргумент `{{title}}`, переданный в шаблон представления для замены параметров.

Наша визуализированная HTML-страница теперь начинает выглядеть как настоящая. Однако нам по-прежнему нужны теги `<doctype>`, `<head>` и `<body>`, чтобы этот код был действительным. Handlebars, как и другие шаблонизаторы, позволяет указать шаблон основного макета, который будет использоваться в качестве основного макета для всех страниц. По умолчанию он называется `layout.hbs`:

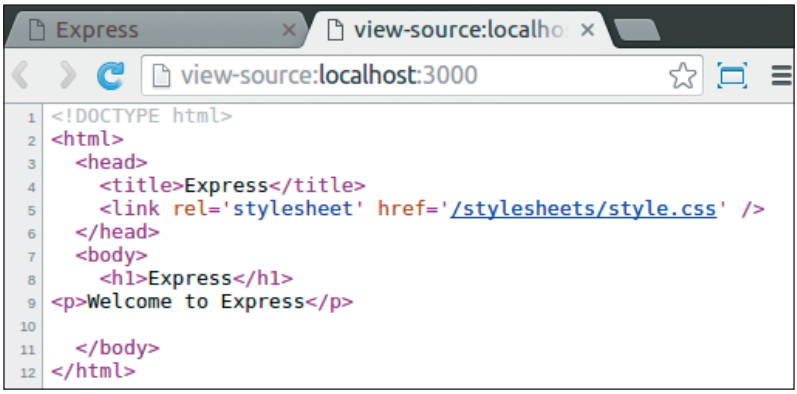
```
<!DOCTYPE html>
   <html>
   <head>
      <title>{{title}}</title>
      <link rel='stylesheet' href='/stylesheets/style.css' />
   </head>
   <body>
      {{{body}}}
   </body>
</html>
```

Здесь мы определили шаблон макета, который будет использоваться для каждого представления. Handlebars создаст HTML-страницы, начинающиеся с это-

го шаблона, а затем заменит любой конкретный шаблон представления в теге `{{{body}}}`. Этот базовый шаблон включил таблицу стилей в тег `<link>` в элемент `<head>`, что и следовало ожидать для стандартной HTML-страницы. Обратите внимание, что элемент `<title>` использует параметр подстановки `{{title}}`. Наш обработчик запроса входа в систему визуализирует эту страницу с объектом, который содержит свойство `title`. Поэтому Handlebars будет использовать этот объект для замены параметра `{{title}}` переданным значением объекта. Наша итоговая страница выглядит так:



А наш исходный HTML-код для этой страницы выглядит так:



События POST

В настоящее время наше приложение Express визуализирует только индексную страницу, используя шаблонизатор Handlebars. Давайте теперь расширим это приложение для визуализации формы входа в систему, а затем обработаем результаты, когда пользователь заполнит эту форму и отправит ее обратно в наше приложение.

Для этого нам понадобится обработчик маршрута входа в систему, который будет принимать как метод `GET`, так и метод POST. Наше приложение нужно будет изменить в нескольких местах. Во-первых, нам понадобится обработчик запроса `GET`, который будет использовать связанный шаблон представления `login.hbs` для визуализации формы входа. Во-вторых, нам понадобится еще один обработчик для обработки запроса POST, как только пользователь заполнит форму и на-

жмет кнопку отправки. Обработчик POST должен будет проанализировать данные POST.

Давайте начнем с представления login.hbs в каталоге views, которое содержит простую HTML-форму:

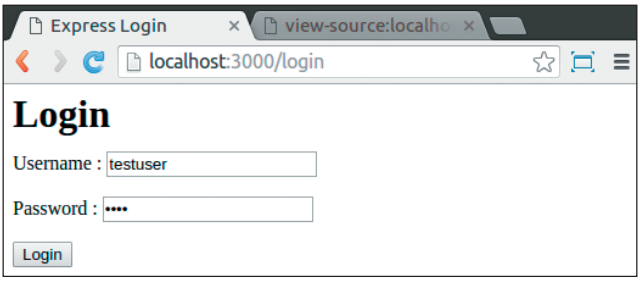
```
<h1>Login</h1>
<p>
   form method="post">
      <p>{{ErrorMessage}}</p>
      <p>Username : <input name="username"></input></p>
      <p>Password : <input name="password"></input></p>
      <button type="submit">Login</button>
   </form>
</p>
```

Здесь мы создали HTML-форму, содержащую несколько стандартных элементов. Для начала у нас есть элемент <p> для отображения свойства представления с именем {{ErrorMessage}}, которое будет использоваться для отображения любых ошибок отправки пользователю. Далее у нас идут два поля ввода, Username и Password, а также кнопка **Login** для отправки формы.

Теперь, когда у нас есть это представление, мы можем обновить наш файл route/Login.ts, чтобы визуализировать это представление по запросу GET:

```
import express from 'express';
var router = express.Router();

router.get('/login', (req: express.Request,
   res: express.Response) => {
      res.render('login',
      {
         title: 'Express Login'
      }
   );
});

export { router } ;
```

Здесь мы изменили наш обработчик маршрутов, чтобы просто визуализировать представление 'login', и установили свойство title как строку, содержащую значение 'Express Login'. Запустив наше приложение сейчас и перейдя по адресу http://localhost: 3000/login, мы вызовем обработчик запроса на вход, и на экране появится простая форма входа в систему:



Теперь, когда форма входа в систему появилась на экране, мы можем сосредоточиться на обработке значений формы после их отправки. Нажатие на кнопку `Login` приведет к тому, что HTML-страница сгенерирует сообщение `POST` для обработчика запроса на вход. Поэтому нам нужно указать обработчик, который подхватит это сообщение. Для этого нам нужно изменить обработчик `Login.ts` и включить новый обработчик `POST`:

```
router.post('/login', (req, res, next) => {
   if (req.body.name.length > 0) {
      req.session!['username'] = req.body.username;
      res.redirect('/');
   } else {
      res.render('login', {
         title: 'Express',
         ErrorMessage: 'Please enter a user name'
      });
   }
});
```

Мы вызываем метод `post` модуля `router`. Как мы видели в случае с функцией `get`, Express использует функцию `post` для настройки обработчика событий `POST` в нашем модуле.

Этот обработчик проверяет свойство `request.body.username` для считывания данных формы из опубликованного запроса формы. Если свойство `username` является допустимым (в данном случае это просто то, что было введено), мы сохраняем значение в свойстве сессии с именем `req.session['username']` и перенаправляем браузер на страницу по умолчанию. Если свойство `username` не было введено, мы просто повторно визуализируем представление входа в систему и сообщение об ошибке.

Однако, прежде чем мы протестируем новую страницу входа в систему, нам нужно будет установить и настроить несколько модулей:

```
npm install body-parser --save
npm install cookie-parser --save
npm install express-session -save
```

Модуль `body-parser` используется для анализа данных формы как результат события `POST` и присоединения этих данных к самому объекту запроса. Это означает, что мы можем просто использовать `req.body` для разыменования данных формы.

Модули `cookie-parser` и `express-session` используются для обработки сеанса. В нашем обработчике `POST` входа в систему мы устанавливаем переменную сеанса для свойства данных формы `username`. Без этих двух модулей это не сработает.

Последнее изменение, которое нам нужно сделать, – это импортировать эти модули в наше приложение и выполнить любую необходимую им конфигурацию. Поэтому нам необходимо обновить файл приложения `app.ts`:

```
// Существующий код
app.set('view engine', 'hbs');

import bodyParser from 'body-parser';
import cookieParser from 'cookie-parser';
import expressSession from 'express-session';

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(expressSession({secret: 'asdf' }));

// Существующий код
app.use('/', Index.router);
```

Здесь мы импортируем новые модули, используя наш стандартный синтаксис модуля `import`. Затем мы выполняем четыре вызова функции `app.use`, чтобы настроить каждый из модулей. Модуль `body-parser` использует вызов функции `json()` для возврата посредников, которые Express будет использовать для преобразования входящих запросов в объекты, прикрепленные к `req.body`. `body-parser` также должен установить свойство `urlencoded`, чтобы сделать JSON-подобные объекты открытыми. Эти две настройки создадут простой Java-объект POJO, доступный через свойство `req.body` при получении запросов `POST`.

Модуль `cookie-parser` настраивается простым использованием экспортированной функции-конструктора, как и модуль `express-session`. Обратите внимание, что модули `cookie-parser` и `express-session` необходимы для хранения переменных в объекте `req.session`.

После этого наш обработчик POST-запросов сможет запросить `req.body.username`, чтобы найти введенное имя пользователя, и `req.body.password`, чтобы найти соответствующий пароль. Он также сможет хранить значения в сеансе.

Перенаправление HTTP-запросов

Теперь, когда у нас есть работающий модуль входа в систему для обработки простого запроса на вход, мы можем перенаправить сеанс браузера обратно на нашу домашнюю страницу и обработчик запросов `Index.ts` через вызов `res.redirect('/')`. Давайте обновим наш обработчик `Index.ts` для работы со значением `username`, которое мы сохранили в сеансе:

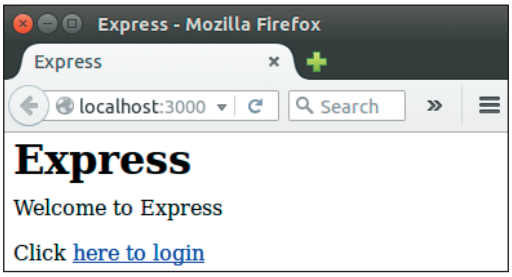
```
router.get('/', (req, res, next) => {
   res.render('index',
      { title: 'Express',
        username: req.session!['username']
      }
   );
});
```

Здесь мы просто добавили новое свойство к объекту, который передается в наш шаблон `index.hbs` с именем username. Значение этого свойства извлекается из нашего сеанса. Теперь мы можем обновить шаблон представления `index.hbs`:

```
<h1>{{title}}</h1>
<p>Welcome to {{title}}</p>

{{#if username}}
   <p>User: {{username}} logged in.
{{else}}
   <p>Click <a href="/login">here to login</a></p>
{{/if}}
```

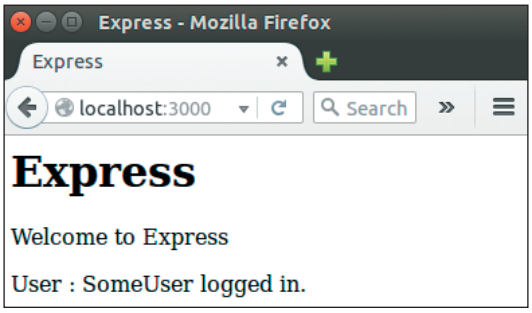
Мы добавили блок кода в наш шаблон Handlebars, который использует логику JavaScript для рендеринга другого HTML-кода на основе свойства `username`. Если свойство `username` имеет значение, то мы показываем, что пользователь вошел в систему. Если нет, мы отображаем ссылку на обработчик запроса `'/login'`, чтобы позволить пользователю войти в систему. Проще простого.

Запустив теперь наше приложение, мы увидим домашнюю страницу со ссылкой для входа:



Перейдя по ссылке для входа, мы увидим экран входа:

После того как мы заполнили форму и нажали на кнопку **Login**, обработчик запроса на вход в систему обработает наш запрос, а затем перенаправит наш браузер на домашнюю страницу:



Обратите внимание, что ссылка **Click here to login** (**Нажмите здесь для входа**) исчезла, согласно нашей логике, и отображается значение имени пользователя (из сеанса), так как пользователь вошел в систему.

В этом разделе мы рассмотрели модуляризацию применительно к Node и движку Express. Мы начали с простого приложения Express и создали простой обработчик запросов в качестве модуля Node. Затем мы исследовали возможности маршрутизации Express и создали два отдельных модуля, один для обработки запросов к нашей главной странице, а другой для обработки функций входа в систему. Далее мы познакомились с шаблонизатором Handlebars и создали три представления: представление `layout.hbs`, которое содержало общую структуру страницы, представление главной страницы и представление страницы входа. После этого мы поработали с обработчиком запросов `POST` и показали, как анализировать значения форм и сохранять свойство в сеансе пользователя. Наконец, мы показали, как работает перенаправление, и связали эти две страницы вместе, чтобы реализовать функциональность входа в систему в своем приложении.

Функции Lambda

В мире веб-приложений Node изменил правила игры. Одной из причин этого являются легкие аппаратные спецификации, необходимые для работы веб-сервера Node.

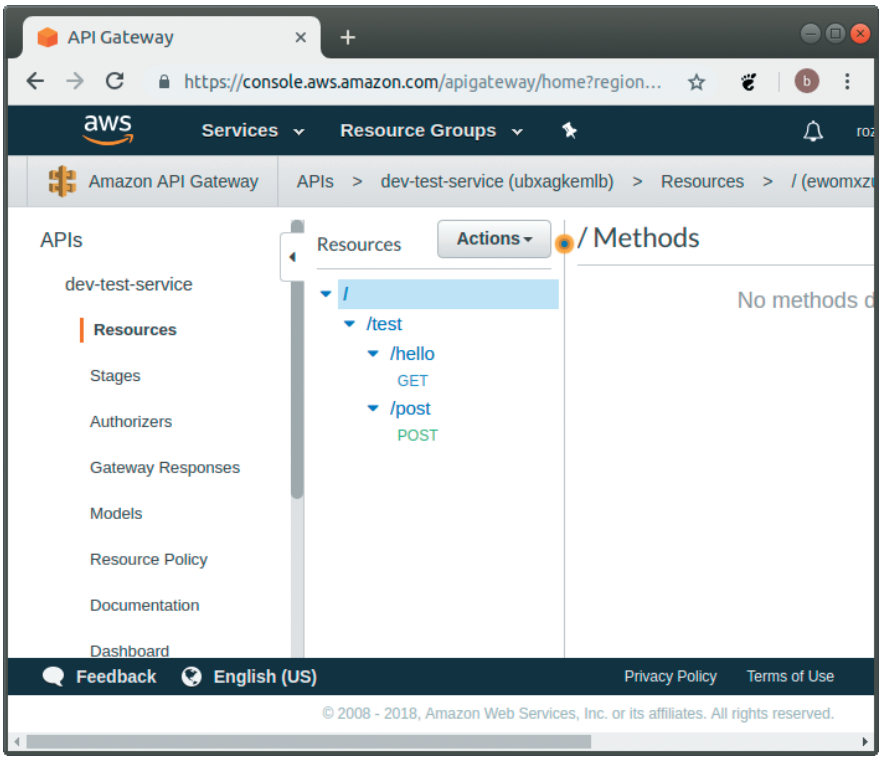
Приложения Node можно запускать практически на любом оборудовании, начиная с чего-то маленького, например Rasberry Pi. Это означает, что стоимость настройки веб-сервера Node значительно снижается, по сравнению со стоимостью сервера, на котором выполняется веб-приложение .NET или Java. В современную эпоху облачных вычислений это означает, что маленький, недорогой облачный сервер можно легко использовать для размещения приложения Node.

Большинство облачных сервисов, в том числе Azure, Amazon и Google, продвинуло эту концепцию на шаг вперед и теперь предлагает возможность запуска кода без необходимости сервера вообще. Это означает, что нам предоставляется среда выполнения, которая имеет все зависимости, необходимые для того, чтобы ответить на веб-запрос. Если мы возьмем Node в качестве примера, до тех пор, пока у нас есть базовые модули Node, доступные нам во время выполнения, чтобы настроить обработчик для ответа на один веб-запрос, требуется всего несколько строк кода. Именно для этой цели поставщики облачных услуг смогли связать все зависимости в миниатюрную среду выполнения. Все, что нам нужно сделать, – это загрузить несколько строк кода для обработки веб-запроса, а поставщик облачных услуг обеспечит наличие необходимой среды для обработки этого запроса.

Microsoft Azure предоставляет так называемые функции Azure для этой цели, а **Amazon Web Services** (**AWS**) предоставляет так называемые функции Lambda. Эти функции могут быть написаны на .NET, Go, Java, Node, Python или даже Ruby. В этом разделе мы рассмотрим настройку простой функции Lambda, чтобы показать, как создавать и использовать модуль Node в одной из этих предварительно сконфигурированных сред выполнения.

Архитектура функции Lambda

Прежде чем мы продолжим и создадим функцию Lambda, давайте сначала рассмотрим три компонента архитектуры AWS, которые работают вместе для ответа на веб-запрос. Первое, что нам нужно определить, – это конечная точка API или, с точки зрения AWS, API-ресурс. У API-ресурса есть путь, который будет соответствовать URL-адресу, и действие. Это может быть `GET`, `POST`, `PUT` или любое другое стандартное действие REST. Например, один запрос GET для URL-адреса `/test/hello` представляет собой один API-ресурс, как можно увидеть на приведенном ниже скриншоте:



Здесь мы видим два API-ресурса. Первый – это запрос `GET` к URL-адресу `/test/hello`, а второй – запрос `POST` к URL-адресу `/test/post`.

Во-вторых, каждый API-ресурс имеет так называемый этап. Этап – это область, где будет развернут этот API, и он позволяет дополнительно настраивать API в зависимости от его предназначения. Например, вы можете настроить этап с именем `dev`, другой с именем `test`, и заключительный этап под названием `prod`, что разрешает развертывание на этапе `dev` или `test` без каких-либо последствий для этапа `prod`. Имя этапа составляет первую часть URL-адреса для любой конкретной конечной точки API. Например, для этапа с именем `dev` полный путь к GET-запросу для API-ресурса будет выглядеть так: `/test/hello is /dev/test/hello`.

Последний архитектурный компонент – сама функция Lambda. API-ресурс можно настроить для выполнения функции. Она будет выполняться в контексте API-ресурса, и ей предоставляется информация об этом контексте при выполнении.

Использование ресурсов и функций Lambda очень похоже на использование обработчика Node Express. Оба имеют путь, который указывает, какой обработчик будет отвечать на запрос, и оба имеют код, который будет выполняться при запросе конечной точки. Однако ресурсы AWS используют этап, который напоминает настройку совершенно нового экземпляра Node Express на другом сервере.

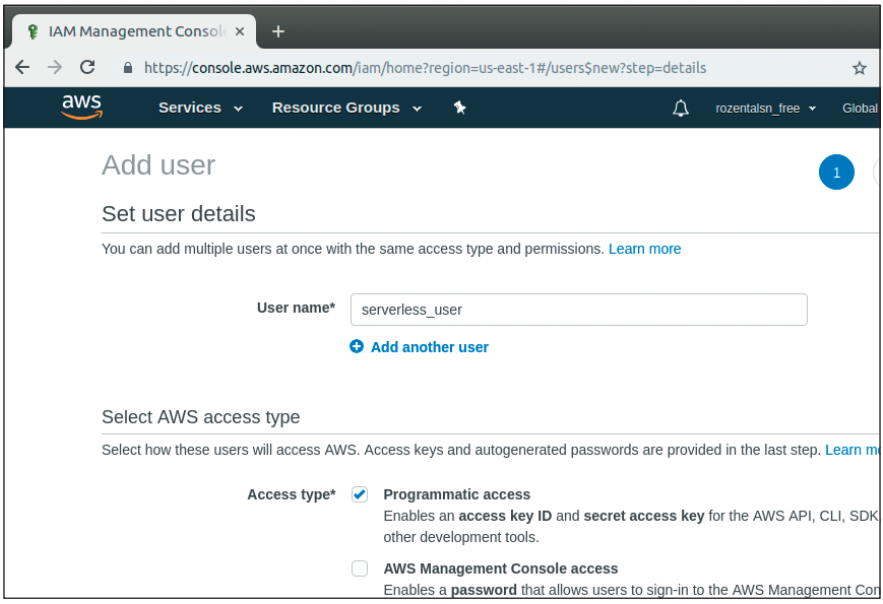
Настройка AWS

Чтобы использовать AWS, нам нужно настроить учетную запись. Мы не будем здесь рассматривать создание учетной записи, но это относительно простой процесс. AWS предоставляет широкий спектр учетных записей и вариантов выставления счетов, а также уровень бесплатного пользования, который не потребует никаких затрат.

В этом уровне есть ограничения, которые очень щедры, поэтому обязательно ознакомьтесь с условиями, чтобы убедиться, что использование вами бесплатного уровня действительно превышает эти лимиты.

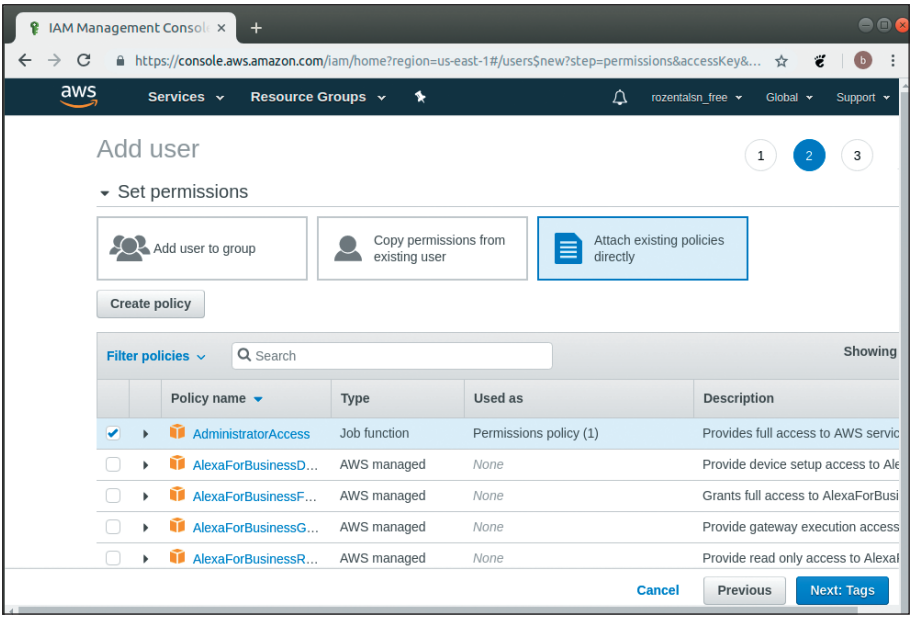
После настройки учетной записи нам нужно будет создать пользователя, у которого есть программный доступ к API-интерфейсу AWS. Этот интерфейс позволяет нам использовать внешние инструменты для создания и управления любыми сервисами, доступными через AWS, включая создание виртуальных машин, настройку групп безопасности и вообще всего, что может сделать интерфейс GUI. Чтобы создать пользователя, откройте раздел управления удостоверениями и доступом, выберите **Users** (**Пользователи**), а затем **Add user** (**Добавить пользователя**).

Добавьте имя своего пользователя и убедитесь, что установлен флажок **Programmatic access** (**Программный доступ**), как показано на приведенном ниже скриншоте:

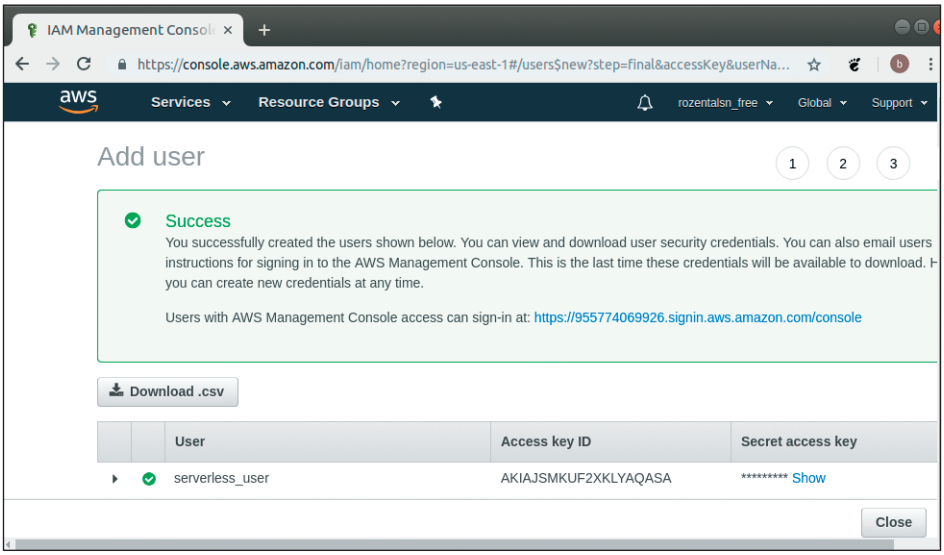


Нажмите кнопку **Next: Permissions** (**Далее: Права доступа**), а затем выберите вкладку **Attach existing policies directly** (**Прикрепить существующие полити-**

ки напрямую), чтобы отобразить список доступных политик. Выберите политику **AdministratorAccess** и нажмите **Next: Tags** (**Далее: Теги**):



Мы не будем создавать никаких тегов для этого пользователя, поэтому нажмите кнопку **Next: Review** (**Далее: Обзор**), а затем – **Create User** (**Создать пользователя**). После этого для нас будет создан пользователь и появится экран **Add user** (**Добавить пользователя**), сообщая об успешном создании пользователя:



Здесь видно, что был успешно создан новый пользователь с именем **serverless_user** и ему был присвоен **Access key ID** (**Идентификатор ключа доступа**), а также **Secret access key** (**Секретный ключ доступа**). Нам понадобятся оба этих ключа, чтобы иметь возможность делать что-либо с помощью AWS API, поэтому обязательно храните их где-нибудь в безопасном месте. Здесь также есть кнопка **Download.csv**, которая позволит нам скачать CSV-файл с этими ключами.

Теперь, когда у нас настроен пользователь, который может использовать AWS API, мы можем создать API-ресурс и присоединить к нему функцию Lambda.

Serverless

Все сервисы AWS, доступные для использования, имеют портал с графическим интерфейсом, который позволяет создавать, удалять, настраивать и управлять всем, что связано с AWS. Хотя это дает возможность изучить множество доступных вариантов и конфигураций, очень легко позабыть, какие именно шаги потребовались для настройки конкретного ресурса. Когда у нас есть большая система для развертывания в конкретной учетной записи AWS, имеет смысл записать эти параметры конфигурации и настройки, чтобы мы могли легко повторять эти шаги снова и снова. Данный процесс называется автоматизированным развертыванием.

Serverless – это пакет автоматического развертывания с открытым исходным кодом, который может работать с несколькими облачными платформами. Его можно использовать для развертывания у нескольких поставщиков, включая AWS, Azure, Google Cloud или даже локальную среду разработки. Он очень гибкий и расширяемый и скрывает большую часть низкоуровневых вызовов API-функций, которые требуются каждому поставщику. В этом разделе мы настроим Serverless для развертывания API-ресурса AWS, подключим его к этапу и загрузим источник функции Lambda с простыми файлами конфигурации. Затем мы напишем простую программу на основе Node для доступа к этим вновь созданным конечным точкам REST.

Serverless, будучи модулем Node, легко устанавливается, если просто запустить следующую команду в командной строке:

```
npm install -g serverless
```

Эта команда установит Serverless в качестве глобального модуля и сделает его доступным для использования из командной строки. После установки мы можем проверить установленную версию, выполнив следующее:

```
serverless -v
```

На момент написания используется Serverless версии 1.34.1:

```
1.34.1
```

Настройка Serverless

Чтобы использовать Serverless с нашей учетной записью AWS, нам нужно будет настроить наши учетные данные в командной строке. Для этого шага потребуется информация, которую мы загрузили при создании нашего пользователя AWS с программным доступом, в частности **идентификатор ключа доступа** API и **секретный ключ доступа**.

Чтобы настроить Serverless с нашими учетными данными AWS, выполните в командной строке следующее:

```
serverless config credentials --provider aws --key <insert
access key> --
secret <insert secret key>
```

Здесь мы указали параметры конфигурации для учетных данных, которые Serverless будет использовать для доступа к API-интерфейсу AWS. В число указанных нами параметров входят aws в качестве поставщика API, а также **идентификатор ключа доступа** API и **секретный ключ доступа**.

После завершения этого шага Serverless сохранит эти учетные данные для последующего использования.

Теперь мы можем создать стандартный шаблон Serverless для функции Lambda, выполнив в командной строке следующее:

```
serverless create --template aws-nodejs --path serverless-sample
```

После этого будет создан новый каталог с именем `serverless-sample`, а внутри него файлы `serverless.yml` и `hander.js`. Давайте сначала посмотрим на сгенерированный файл `hander.js`:

```
'use strict';

module.exports.hello = async (event, context) => {
  return {
    statusCode: 200,
    body: JSON.stringify({
      message: 'Go Serverless v1.0! Your function executed
        successfully!',
      input: event,
    }),
  };
};
```

Здесь у нас есть файл JavaScript, который использует свойство `module.exports` для предоставления функции `hello`, как видно из использования `module.exports.hello`. Эта функция и есть сама функция Lambda, которая определя-

ется как асинхронная функция с двумя параметрами, `event` и `context`. Эти параметры заполняются AWS для передачи информации из API-ресурса в функцию Lambda. Тело функции возвращает объект JavaScript, который имеет свойства `statusCode` и `body`. Этот объект представляет собой стандартный HTTP-ответ, а свойство `statusCode` является возвращаемым HTTP-кодом. В этом примере HTTP-запрос вернет ответ `200, OK`. Свойство `body` содержит свойство `message`, а также свойство `input`.

Обратите внимание, насколько близко этот код напоминает обработчик Express. Помимо параметров события и контекста, и обработчики Express, и наши обработчики AWS Lambda прикрепляют функцию к `module.exports`, и оба возвращают HTTP-ответы.

Развертывание

Теперь, когда у нас есть функция Lambda, нам нужно присоединить ее к ресурсу AWS, для которого нам нужно указать свойства `stage` и `path`. Serverless создал файл `serverless.yml`, который содержит эту необходимую информацию. Давайте посмотрим на него:

```
service: serverless-sample

provider:
  name: aws
    runtime: nodejs8.10

functions:
  hello:
    handler: handler.hello
```

Здесь в нашем `.yml` или YAML-файле есть ряд свойств. Обратите внимание, что YAML использует отступ для упорядочения свойств, поэтому свойства верхнего уровня идут ровно, а дочерние свойства имеют отступ. Это означает, что свойства `service`, `provider` и `functions` находятся на одном уровне, а у свойства `provider` есть два дочерних свойства, `name` и `runtime`. Следуя этому шаблону отступов, у свойства `functions` есть дочернее свойство `hello`, у которого, в свою очередь, есть дочернее свойство `handler`.

Свойства в этом файле указывают, что имя службы будет `serverless-sample`, а поставщиком служб будет AWS. Кроме того, механизмом исполнения, который будет использоваться для функции Lambda, будет версия NodeJ 8.10. Свойство `functions` обеспечивает имя функции, и, наконец, свойство `handler` указывает используемый файл JavaScript, который в этом случае называется `handler`, за которым следует точка, а затем имя экспортируемой функции, `hello`.

Однако в этом файле отсутствует ряд вещей. Нам все еще нужно указать путь к конечной точке REST, а также HTML-метод, `GET`. Мы также должны указать этап и область AWS, в которые должна быть помещена эта функция Lambda. В документации к Serverless объяснено, где эти свойства должны быть размещены. Она есть на сайте `docs.serverless.com`. Давайте продолжим и изменим этот файл:

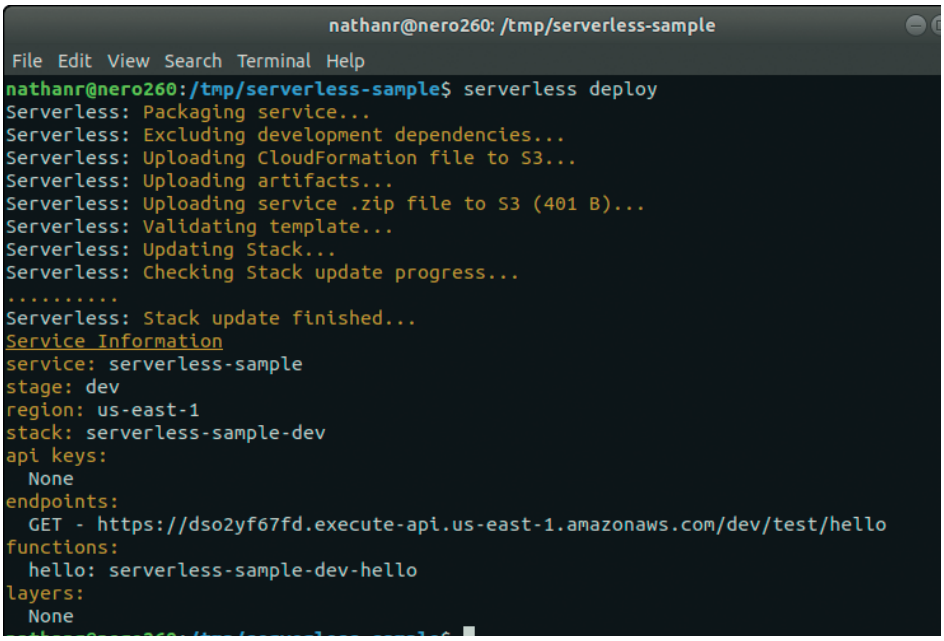
```
service: serverless-sample

provider:
  name: aws
  runtime: nodejs8.10
  stage: dev
  region: us-east-1

functions:
  hello:
    handler: handler.hello
    events:
      - http:
        path: test/hello
        method: get
```

Здесь мы сделали две вещи. Во-первых, мы добавили свойства `stage` и `region`, которые будут развертывать нашу функцию в этап `dev` и область `us-east-1`. Мы также добавили свойство `events` на том же уровне, что и существующее свойство `handler`, и добавили свойство `http` со свойством `path`, указывающее путь `test/hello`, и свойство `method`, указывающее метод `get`. Обратите внимание, что перед свойством `http` установлена черточка (`-`).

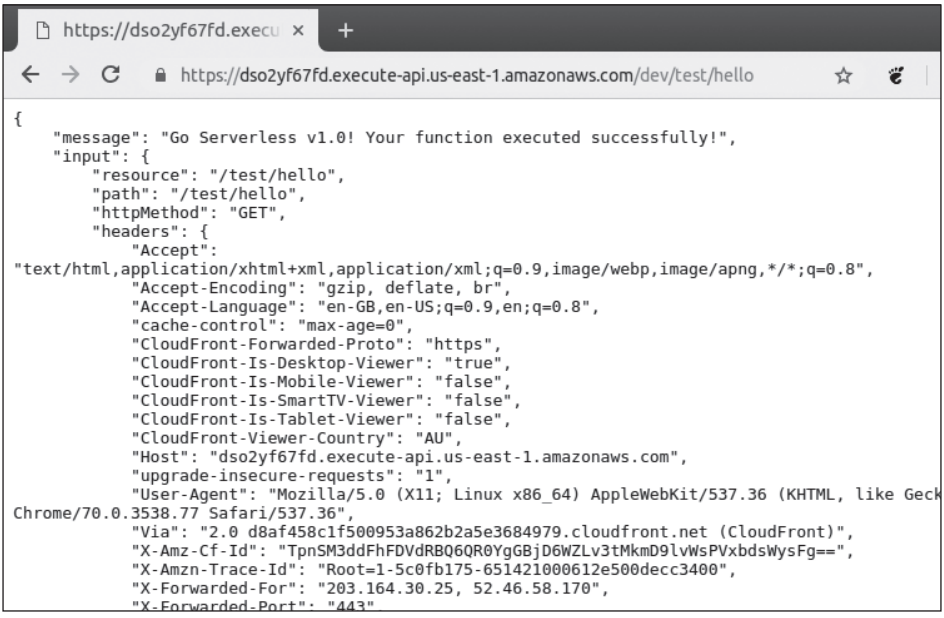
После внесения этих незначительных изменений мы можем развернуть нашу функцию Lambda, выполнив в командной строке следующее:

```
serverless deploy
```

После этого Serverless упакует функцию и развернет ее в нашей учетной записи AWS, используя настроенного нами ранее пользователя. Если все пойдет хорошо, Serverless запишет в консоль полный URL-адрес нашей вновь созданной функции Lambda, как показано ниже:



Наша функция была успешно развернута, и Serverless сообщил полный URL-адрес в консоли. Если мы скопируем и вставим это значение в браузер, то увидим, что функция ответила на наш запрос GET и вернула довольно большой объект JSON, как показано далее:



Функции Lambda в TypeScript

Пример функции Lambda, которую создал для нас Serverless, находится в стандартном JavaScript-файле `handler.js`. Давайте теперь посмотрим, как можно использовать TypeScript для создания совместимого файла JavaScript, который можно использовать в качестве функции Lambda. Для начала давайте инициализируем npm в этом каталоге:

```
npm init
```

С помощью данной команды мы создадим файл `packages.json` в каталоге `serverless-sample`, чтобы иметь возможность использовать его для установки любых модулей, которые нам могут понадобиться. Затем мы можем инициализировать TypeScript в этом каталоге, как обычно:

```
tsc --init
```

После этого будет создан файл `tsconfig.json`, который готов для компиляции TypeScript. Чтобы использовать функции Lambda в TypeScript, нам нужно установить два пакета:

```
npm install aws-lambda --save
npm install @types/aws-lambda -saveDev
```

Перед созданием функции Lambda в TypeScript нам нужно обновить файл `tsconfig.json`, чтобы он включал в себя стандартные библиотеки `es2015` и `dom`:

```
"module": "commonjs",
"lib": [
   "es2015",
   "dom"
],
```

Записи `lib` позволят нам использовать промисы в нашем коде.

Давайте теперь создадим файл tshandler.ts со следующим содержимым:

```
import { APIGatewayProxyEvent,
   APIGatewayProxyResult,
   Handler,
   Context } from 'aws-lambda';

export const handler: Handler =
   (event: APIGatewayProxyEvent, context: Context):
      Promise<APIGatewayProxyResult> => {
         return new Promise<APIGatewayProxyResult>((resolve,
            eject) => {
            console.log(`event : ${JSON.stringify(event, null, 4)}`);
```

```
            resolve({
               statusCode: 200,
               body: JSON.stringify({
               message: 'TypeScript Serverless v1.0!',
               input: event,
            }, null, 4)
         });
      })
   }
```

Вначале идет импорт нескольких типов из модуля `aws-lambda`. Затем мы экспортируем постоянную функцию `handler`, типа `Handler`. Этот тип гарантирует, что мы определяем функцию с параметрами `event` и `context` и что функция возвращает промис типа `APIGateWayProxyResult`.

Как видно, просто преобразовав наш код из JavaScript в TypeScript, мы можем использовать строгую типизацию и понимать входные и выходные данные каждой из наших функций, как определено в файле объявлений. Функция handler просто возвращает новый промис и дублирует HTTP-ответ `200`, как это было в версии JavaScript.

Давайте теперь обновим файл `serverless.yml` для развертывания этой функции Lambda в новой конечной точке:

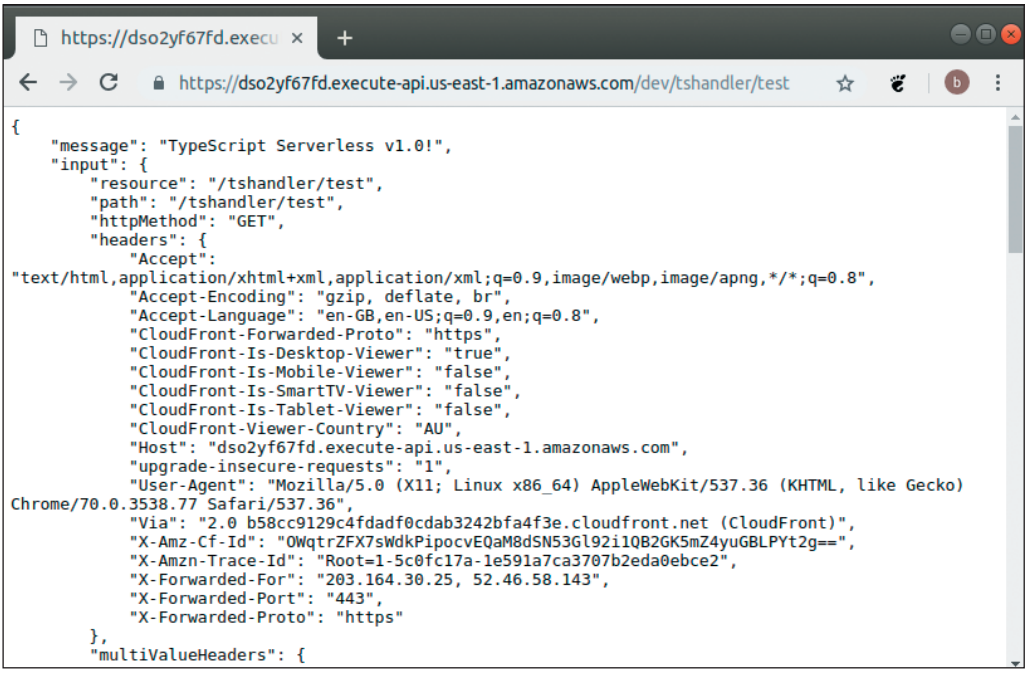
```
functions:
  hello:
    (... existing )
  tslambda:
    handler: tshandler.handler
    events:
      - http:
        path: tshandler/test
        method: get
```

Здесь мы создали новое свойство под свойством `functions`, которое находится на том же уровне, что и существующее свойство `hello`, `tslambda`. Свойство `handler` состоит из имени файла JavaScript, без расширения `.js`, и имени функции, которая была экспортирована, а именно `tshandler.handler`. Свойство `events` теперь указывает, что URL-путь к нашей новой функции Lambda в TypeScript – это `tshandler/test`, и это для запроса `GET`. Как только мы развернем это, используя `serverless deploy`, новая конечная точка REST будет показана в консоли:

```
https://dso2yf67fd.execute-api.us-east-1.amazonaws.com/dev/
tshandler/test
```

Как видно, URL-адрес включает в себя свойство этапа dev, а также путь `tshandler/test`, который мы указали. Если мы запустим браузер и скопируем этот URL-

адрес, то увидим, что наша новая функция Lambda возвращает аналогичный JSON-пакет в образец Serverless:



Мы только что успешно скомпилировали функцию Lambda в TypeScript и развернули ее в нашей учетной записи AWS.

Node-модули функции Lambda

В любой обычной разработке, основанной на Node, мы можем использовать множество модулей, которые были написаны и сделаны доступными как проекты с открытым исходным кодом. До сих пор в этой главе мы использовали такие модули, как `Express`, `cookie-parser`, `lodash` и даже `Jasmine` и `SystemJS`.

При разработке функций Lambda мы также можем использовать любой модуль Node, который нам нужен, если он включен во время нашего развертывания. Во многом так же, как мы должны выполнить команду npm `install <имя_модуля>`, чтобы загрузить модуль в наш каталог `node_modules`, мы должны иметь возможность включать эти модули во время выполнения функции Lambda. К счастью, процесс включения модулей Node очень прост и основан на стандартном механизме npm, к которому мы привыкли.

В качестве примера того, как включить внешние node-модули в функцию Lambda, давайте создадим обработчик событий POST, который будет принимать дан-

ные в этой функции. Этот обработчик POST будет включать в себя модуль http-status-codes, который включает в себя понятные человеку определения всех кодов состояния HTTP, которые, возможно, потребуется вернуть функции Lambda. Установка осуществляется через npm:

```
npm install http-status-codes
```

Обратите внимание, что модуль npm http-status-codes следует современной тенденции включения файла определения TypeScript как части самого модуля, а это означает, что нам не нужно устанавливать отдельный пакет @types только для того, чтобы разрешить интеграцию TypeScript. Установив этот пакет, мы можем создать новый файл с именем posthandler.ts:

```
import { APIGatewayProxyEvent,
  APIGatewayProxyResult,
  Handler,
  Context } from 'aws-lambda';
import * as HttpStatusCodes from 'http-status-codes';

export const postHandler: Handler =
  (event: APIGatewayProxyEvent, context: Context):
    Promise<APIGatewayProxyResult> => {
    return new Promise<APIGatewayProxyResult>(
      (resolve, reject) => {
        console.log(`event.body :
          {JSON.stringify(event.body)}`);
        resolve({
          statusCode: HttpStatusCodes.OK,
          body: JSON.stringify({
            message: 'TypeScript Serverless POST
              executed successfully!',
            bodyReceived: event.body
          }, null, 4)
        })
      })
    }
```

Здесь мы импортировали необходимые модули из aws-lambda, а также статический класс HttpStatusCodes из модуля http-status-codes. Наша экспортируемая функция носит имя postHandler и возвращает промис, как мы видели ранее. В этой функции есть два заметных изменения. Во-первых, мы записываем в консоль сообщение, содержащее свойства объекта event.body. Это похоже на то, как мы использовали модуль Express body-parser для автоматического анализа тела HTTP-запроса POST и предоставления нам доступа к базовому объекту. Поэтому наш код может обращаться к любому из параметров POST таким способом.

Второе заметное изменение заключается в использовании статического перечисления `HttpStatusCodes.OK` в функции `resolve`. Это перечисление является частью модуля `http-status-codes` и позволяет нашему коду использовать удобочитаемые и понятные значения для стандартных кодов HTTP-ответов.

Свойство `body` этой функции Lambda просто возвращает JSON-объект, который выводит полученные аргументы `event.body`, чтобы мы могли проверить, что то, что мы отправили, было получено правильно.

Теперь мы можем обновить файл `serverless.yml`:

```
provider:
  (... existing)

package:
  include:
    - ../node_modules/http-status-codes/**

functions:
  hello:
    (... existing)
  tsLambda:
    (... existing)
  posthandler:
    handler: posthandler.postHandler
    events:
      - http:
        path: posthandler/post
        method: post
```

Здесь мы включили новое свойство верхнего уровня с именем `package`, а под ним – свойство `include`, которое обращается к каталогу `node_modules/http-status-codes`. Так мы можем включить внешний node-модуль в качестве зависимости для нашей функции Lambda.

Мы также добавили свойство posthandler под свойством `functions`, которое задает все параметры, необходимые нам для развертывания функции `postHandler`. Ее развертывание в нашей учетной записи AWS создаст конечную точку REST, которая будет обрабатывать событие POST по URL-адресу `/dev/posthandler/post`.

Логирование

Код функции `postHandler` включает в себя вызов функции `console.log` для записи значения параметра `event.body`. Если бы это было стандартное консольное приложение Node, мы могли бы просто отслеживать командную строку,

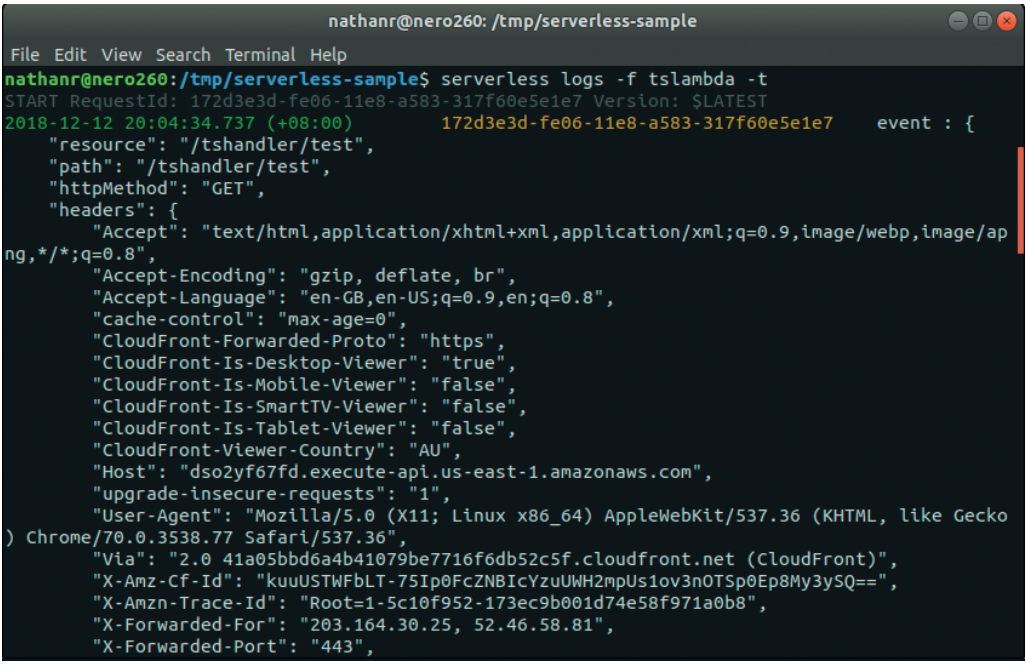
чтобы увидеть результаты этого вызова. К сожалению, так как этот код является функцией Lambda, нам нужно будет найти другой способ доступа к журналам, которые создаются при ее выполнении. Как обычно, веб-интерфейс AWS предоставляет механизм для просмотра этих журналов, но поиск и постоянное обновление веб-страницы, чтобы увидеть последние результаты, занимает довольно много времени. К счастью, Serverless предоставляет простой интерфейс командной строки, который использует API-интерфейс AWS для мониторинга журналов консоли в экземпляре облака AWS и создания отчетов о них в командной строке. Мы можем отслеживать любые журналы событий функции Lambda, просто выполнив следующую команду:

```
serverless logs -f <lambda_name> -t
```

В качестве примера давайте посмотрим на журналы нашего обработчика `GET`:

```
serverless logs -f tslambda -t
```

Вывод этой команды можно увидеть на приведенном ниже скриншоте:



Тестирование REST

До сих пор мы создали обработчик `GET`-запросов функций Lambda и обработчик `POST`. `GET`-запросы довольно просты для тестирования, поскольку мы можем просто скопировать и вставить URL-адрес в веб-браузер для выполнения `GET`-за-

проса. Обработчикам POST, однако, нужен инструмент, такой как, например, Postman, чтобы создать правильный пакет данных для отправки на сервер. Для этого упражнения давайте воспользуемся очень простой программой на основе Node, которая отправит GET-запрос нашему обработчику TypeScript, а затем выдаст POST-запрос.

При разработке крупномасштабных приложений, использующих конечные точки REST, нам часто нужно использовать результаты REST-запроса в качестве входных данных для другого REST-запроса. В качестве примера рассмотрим конечную точку аутентификации, которая принимает имя пользователя и пароль и возвращает зашифрованный токен. Этот токен затем отправляется с каждым последующим REST-запросом, чтобы сервер мог проверить личность пользователя.

Чтобы связать REST-запросы, мы можем легко использовать возможности async await. В дополнение к этому мы также можем использовать любое количество node-модулей, которые предоставляют напоминающий промисы интерфейс для REST-запросов. Одна из таких библиотек называется Axios. Ее можно установить, как обычно, с помощью npm:

```
npm install axios
```

Наша тестовая программа test_rest.ts выглядит так:

```
import axios, { AxiosResponse } from 'axios';

async function run() {
   console.log(`get request`);

   let getResult = <AxiosResponse>await axios({
      method: 'GET',
      url: `https://2yf35yfd.execute-api.us-east-1.amazonaws.com
         /dev/tshandler/test`
   }).catch((err) => {
      console.log(`err: ${err}`);
   });
   console.log(`getResult: ${JSON.stringify(getResult.data,
      null, 4)}`);
}

run();
```

Вначале идет импорт глобального имени функции axios, а также типа AxiosResponse из модуля axios. Затем мы определяем асинхронную функцию с именем run. Эта функция выполняется в нижней части кода и записывает сообщение в консоль, а затем вызывает функцию axios с объектом, у которого есть свойства method и URL. Эта функция помечена как асинхронная, поэтому выполнение кода будет ожидать результата данного запроса, прежде чем продол-

жить. Результат присваивается внутренней переменной `getResult`. Свойство `method` указывает, что это `GET`-запрос, а свойство `url` – это полный URL-адрес нашей функции Lambda. На случай, если произойдет ошибка, у нас есть обработчик `catch`. Затем код записывает результат `GET`-запроса в консоль.

Теперь мы можем запустить эту тестовую программу (после ее компиляции с помощью tsc), выполнив ее в командной строке:

```
node test_rest
```

Вывод будет соответствовать тому, что мы видели при тестировании этого GET-запроса в браузере:

```
get request
getResult : {
   "message": "TypeScript Serverless v1.0!",
   "input": {
      "resource": "/tshandler/test",
      "path": "/tshandler/test",
      "httpMethod": "GET",
...
```

Теперь мы можем обновить наш тестовый код для выдачи `POST`-запроса:

```
...существующий код
console.log(`posting`);

let postResult = <AxiosResponse>await axios.post(
   `https://2yf35yfd.execute-api.us-east-1.amazonaws.com/dev/
     posthandler/post`,
        { testValue: 1, testStringValue: "testString" })
        .catch((err) => {
           console.log(`err: ${err}`);
        });

console.log(`postResult: ${JSON.stringify(postResult.data,
   null, 4)}`);
```

Здесь мы создали переменную с именем `postResult`, которая будет содержать результат `POST`-запроса. Обратите внимание, что мы используем одну из служебных функций `Axios` под названием `post`. Эта функция является упрощенной версией вызова функции `axios`, которую мы использовали ранее, и присоединяет все правильные свойства, которые нужны `POST`. Например, нам больше не нужно указывать свойство `POST method` при использовании этой функции.

Функция `post` может быть вызвана со строкой в качестве первого аргумента и объектом в качестве второго аргумента. Эти аргументы соответствуют свойствам

`url` и `data`, что может быть ясно видно, если мы перейдем к определению данной функции. В нашем коде мы указываем полный URL-адрес функции `POST` и присоединяем объект со свойствами `testValue` и `testStringValue`. Как обычно, у нас есть обработчик `catch` на случай, если что-то пойдет не так, и мы записываем возвращенный результат в консоль.

Выполнение данного кода для функции `POST` покажет следующие результаты:

```
posting
postResult : {
   "message": "TypeScript Serverless POST executed successfully!",
   "bodyReceived": "{\"testValue\":1,\"testStringValue\":
      \"testString\"}"
}
```

Как видно, наша функция работает правильно и успешно анализирует данные `POST` через свойство `event.body`.

Резюме

В этой главе мы рассмотрели использование модулей CommonJS и AMD. Мы изучили синтаксис, используемый для модуляризации, и показали, как экспортировать и импортировать модули. Затем мы исследовали использование синтаксиса модуля AMD с помощью библиотеки Require и обсудили, как позаботиться о зависимостях модуля. После этого мы исследовали использование синтаксиса модуля CommonJS и показали эквивалентную структуру для зависимостей модуля с использованием SystemJS. Наше путешествие продолжилось углубленной дискуссией по модулям Node и Express, где мы собрали воедино приложение для визуализации индексной страницы и страницы входа в систему и обработали входы в систему с помощью информации о сеансах. Затем мы обсудили, как настроить и использовать облачную среду выполнения для нашего кода без необходимости использования какого-либо сервера, а также рассмотрели Amazon Web Services и функции Lambda.

В следующей главе мы рассмотрим принципы объектно-ориентированного программирования и некоторые полезные шаблоны проектирования.

Глава 11

Объектно-ориентированное программирование

В 1995 году **«Банда четырех»** опубликовала книгу *«Приёмы объектно-ориентированного проектирования. Паттерны проектирования»*. В ней ее авторы Эрих Гамма, Ричард Хелм, Ральф Джонсон и Джон Влиссидес описывают ряд классических шаблонов проектирования программного обеспечения, которые представляют простые и элегантные решения типичных проблем программного обеспечения. Если вы никогда не слышали о шаблонах проектирования, таких как шаблон Factory, Composite, Observer или Singleton, то настоятельно рекомендуем прочитать эту книгу.

Шаблоны проектирования, представленные «Бандой четырех», были воспроизведены во многих языках программирования, включая Java и C#. Вилик Вейн написал книгу под названием *«Шаблоны проектирования TypeScript»*, в которой каждый из этих шаблонов реализуется и обсуждается с точки зрения TypeScript.

В `главе 3` *«Интерфейсы, классы и наследование»* мы потратили некоторое время на создание реализации классического шаблона Factory, который является одним из наиболее популярных шаблонов проектирования, описанных «Бандой четырех». TypeScript с его языковыми конструкциями, совместимыми с ES6 и ES7, является прекрасным примером объектно-ориентированного языка. Имея в своем распоряжении классы, абстрактные классы, интерфейсы, наследование и обобщения приложения, TypeScript теперь может в полной мере использовать любые из этих шаблонов проектирования.

Описание реализации каждого из этих шаблонов «Банды четырех» на языке TypeScript – тема, которую нельзя охватить в одной главе, и это будет несправедливо по отношению к превосходному изложению шаблонов, предложенному Виликом Вейном.

Поэтому в этой главе мы сосредоточимся на процессе написания объектно-ориентированного кода и проработаем пример двух шаблонов проектирования, которые очень хорошо работают в связке при работе со сложными макетами пользовательского интерфейса. Это шаблоны проектирования **State** и **Mediator**, которые ориентированы на состояние приложения и на то, как объекты взаимодействуют друг с другом. Мы создадим приложение Angular 2, которое использует довольно сложный дизайн пользовательского интерфейса и включает в себя ряд сложных

анимированных CSS-переходов. Затем мы начнем процесс доработки нашего исходного приложения с целью применения принципов объектно-ориентированного проектирования и обсудим, как взаимодействуют объекты в нашем приложении. После этого мы реализуем шаблоны проектирования State и Mediator, чтобы инкапсулировать логику, которая используется для определения того, какие элементы пользовательского интерфейса отображаются в зависимости от состояния приложения.

В этой главе мы рассмотрим следующие темы:

- принципы объектно-ориентированного программирования;
- использование интерфейсов;
- принципы SOLID;
- проектирование пользовательского интерфейса;
- шаблон State;
- шаблон Mediator;
- модульный код.

Принципы объектно-ориентированного программирования

Любое приложение, которое мы создаем, должно оцениваться с точки зрения объектно-ориентированного передового опыта.

Роберт Мартин опубликовал то, что известно как принципы проектирования SOLID. SOLID – это акроним для пяти различных принципов объектно-ориентированного программирования и проектирования. Следование этим принципам поможет гарантировать, что код, который мы пишем, прост в обслуживании и расширении, легок для понимания и устойчив к изменениям. В нашем нынешнем быстро меняющемся мире мы, как правило, не можем позволить себе тратить огромное количество времени на изменение своих приложений, чтобы соответствовать постоянно меняющимся требованиям. Чем быстрее мы сможем предоставлять обновления для удовлетворения наших бизнес-потребностей, тем больше у нас шансов опередить наших конкурентов. Придерживаясь принципов SOLID, мы получаем хорошую основу, которая позволяет легко вносить изменения в существующий код, чтобы удовлетворить эти быстро меняющиеся требования к нашей кодовой базе.

Программирование в соответствии с интерфейсом

Одним из основных понятий, которых придерживается Банда четырех, является идея, что программисты должны программировать в соответствии *с интерфейсом, а не с реализацией*. Это означает, что программы создаются с использованием

интерфейсов в качестве определенного взаимодействия между объектами. Когда мы программируем в соответствии с интерфейсом, клиентские объекты не знают внутренней логики своих зависимых объектов и поэтому гораздо более устойчивы к изменениям. Определяя интерфейс, мы начинаем закреплять API, который описывает, какую функциональность предоставляет объект, как его следует использовать, а также как несколько объектов взаимодействуют друг с другом.

Принципы SOLID

Расширение программы до принципа интерфейса – это то, что было придумано как принципы проектирования SOLID, которые основаны на идеях Роберта Мартина. Это аббревиатура для пяти различных принципов:

- единственная ответственность;
- открытость/закрытость;
- принцип подстановки Барбары Лисков;
- разделение интерфейса;
- инверсия зависимостей.

Принципы проектирования SOLID заслуживают упоминания всякий раз, когда обсуждается объектно-ориентированное программирование. Давайте кратко рассмотрим каждый из них.

Единственная ответственность

Идея, лежащая в основе принципа единой ответственности, заключается в том, что объект должен иметь только одну ответственность. *Делай что-то одно, и делай это хорошо*. Мы видели примеры этого принципа в различных совместимых с TypeScript фреймворках, с которыми мы работали. Например, класс модели используется для представления одной модели. Класс коллекции используется для представления коллекции этих моделей, а класс представления – для визуализации моделей или коллекций.

Если какой-либо из наших классов становится суперклассом, другими словами, делает множество разных типов вещей, это свидетельствует о том, что мы нарушаем данный принцип. В качестве простого примера, если ваш файл с исходным кодом для определенного класса становится очень длинным, то этот класс, возможно, делает слишком много. Подумайте о том, какова основная ответственность этого класса, а затем разбейте функциональность класса на более мелкие классы.

Открытость/закрытость

Принцип открытости-закрытости гласит, что объект должен быть открыт для расширения, но закрыт для модификации. Другими словами, после того как интер-

фейс был разработан для класса, изменения со временем этого интерфейса должны быть достигнуты посредством наследования, а не путем непосредственного изменения интерфейса.

Обратите внимание, что если вы пишете библиотеки, которые используются сторонними разработчиками через API, то этот принцип важен для разработки API. API всегда должен пытаться обеспечить обратную совместимость, поэтому изменения должны вноситься только через новый релиз.

Принцип подстановки Барбары Лисков

Принцип подстановки Барбары Лисков гласит, что если один объект является производным от другого, то эти объекты можно заменять друг другом, не нарушая функциональности. Хотя этот принцип кажется довольно простым для реализации, он может стать довольно сложным, когда имеешь дело с правилами типизации, связанными с иерархиями расширенных классов, такими как списки объектов или действия над объектами, которые обычно встречаются в коде, использующем обобщения. В этих случаях вводится понятие дисперсии, и объекты могут быть ковариантными, противоположными или инвариантными. Мы не будем здесь обсуждать более тонкие различия, но помните об этом принципе при написании библиотек или кода с использованием обобщений.

Разделение интерфейса

Идея тут состоит в том, что много интерфейсов лучше, чем один универсальный интерфейс. Если мы свяжем этот принцип с принципом единственной ответственности, то начнем смотреть на наши интерфейсы с точки зрения маленьких фрагментов головоломки, работающих вместе, а не как на интерфейсы, охватывающие большие части функциональности.

Инверсия зависимостей

Эта идея утверждает, что мы должны зависеть от абстракций (или интерфейсов), а не от экземпляров конкретных объектов. Опять же, это тот же принцип, что и программирование в соответствии с интерфейсом, а не с реализацией.

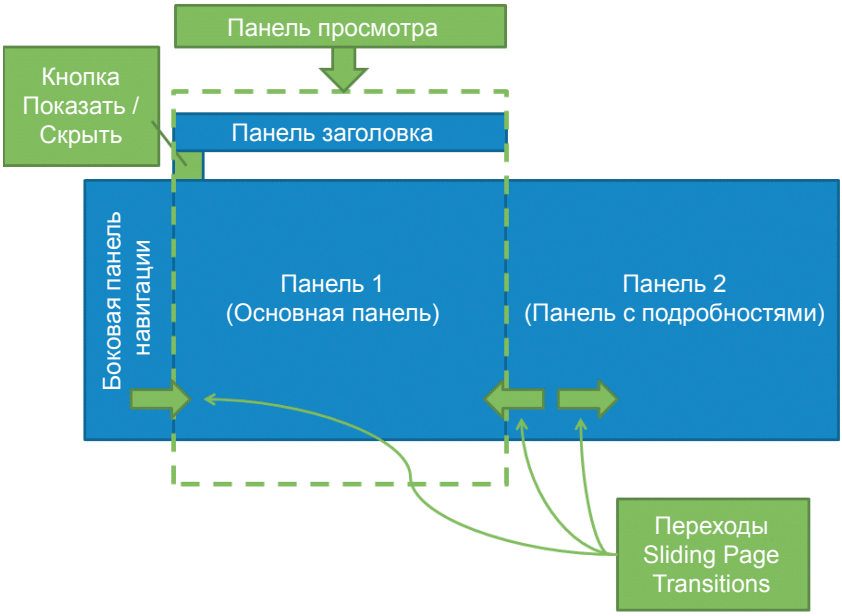
Проектирование пользовательского интерфейса

В качестве примера использования принципов проектирования SOLID давайте создадим приложение, которое использует сложный дизайн пользовательского интерфейса, и посмотрим, как эти принципы могут помочь нам разбить наш код на более мелкие, управляемые модули, разделенные интерфейсами.

В этом разделе мы создадим приложение Angular, которое будет предоставлять макет страницы в стиле панели, перемещающейся слева направо. Мы будем использовать Bootstrap для оформления стилей и CSS-переходы для реализации перемещения панелей слева или справа. Это обеспечит пользователю несколько отличную навигацию по сравнению с обычным дизайном прокрутки вверх-вниз, который используется большинством веб-сайтов.

Концептуальный дизайн

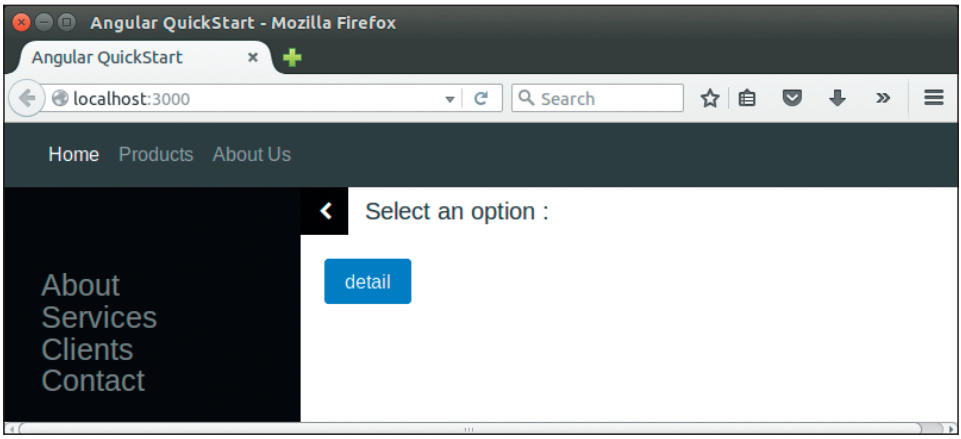
Давайте посмотрим, как концептуально будет выглядеть этот дизайн слева направо:



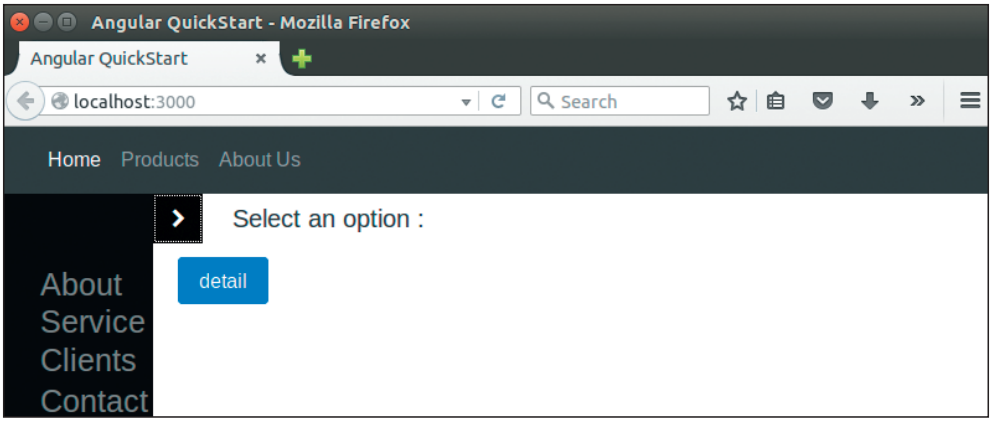
Панель просмотра будет нашей главной страницей с **панелью заголовка** и **кнопкой** для управления отображением или скрытием боковой панели навигации с левой стороны. Когда левая панель открыта, она будет использовать CSS-анимацию, чтобы скользить вправо.

Когда она закрыта, она снова будет использовать анимацию, чтобы скользить назад влево. Аналогичным образом, при нажатии кнопки, чтобы отобразить вторую панель (**панель 2**), эта панель сведений будет скользить влево с использованием CSS-анимации и в конечном итоге займет всю панель просмотра.

На приведенном ниже скриншоте показана **панель просмотра** с **панелью заголовка** и левой боковой панелью:

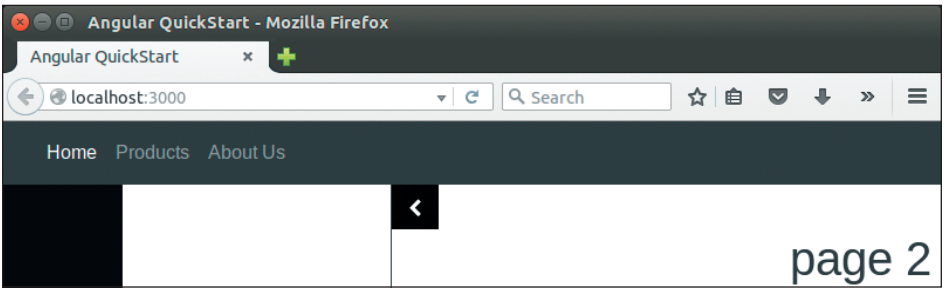


Здесь отчетливо видна панель заголовка вверху, панель меню слева и две кнопки. Первая кнопка находится слева от фразы **Select an option**:. Это просто стрелка <, с помощью которой будет скрыта боковая панель слева. При нажатии на эту кнопку вступает в действие CSS-анимация, чтобы эта боковая панель сдвинулась влево и не мешала. Это можно увидеть при анимированном переходе панели:



Здесь мы приостановили CSS-анимацию, чтобы показать, что левая боковая панель находится в процессе свертывания, а основная панель расширяется, чтобы заполнить всю панель просмотра. Обратите внимание, что стрелка на кнопке **Показать/Скрыть** поменялась на >. Это изменение указывает пользователю на то, что боковую панель можно развернуть, нажав кнопку >.

Если нажать кнопку **detail**, это приведет к тому, что боковая панель слева и главная страница сместятся влево, открывая вторую страницу с помощью другой CSS-анимации, что можно увидеть при анимированном переходе страницы для правой панели:



Здесь вторая страница переходит с правой стороны, а боковая панель слева и главная страница сдвигаются влево.

Настройка Angular

Теперь, когда у нас есть концептуальное представление о том, как будет выглядеть наше приложение, мы можем приступить к реализации этого макета, настроив приложение Angular. Как мы видели в предыдущих проектах Angular, мы начинаем с ввода команды ng new для использования интерфейса командной строки Angular. В ходе этого процесса будут установлены все необходимые зависимости, которые нужны Angular, и будут созданы исходные файлы `app.component.ts` и `app.component.html`.

Файл `app/app.component.ts` очень простой:

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

@Component( {
   selector: 'app-root',
   templateUrl: 'app/app.component.html',
   styleUrls: ['app/app.component.css']
})
export class AppComponent
{
   title = "angular-sample";
}
```

Здесь мы импортируем модуль `Component`, как мы видели ранее, а затем определяем три свойства для передачи декоратору `@Component`. Однако обратите внимание, что вместо указания свойства `template`, которое обычно содержит HTML-код, мы указали свойство `templateUrl`. Оно указывает Angular загружать именованный файл с диска и использовать его в качестве шаблона компонента. Точно так же мы указали CSS-файл, который будет использоваться нашим компонентом с помощью свойства `styleUrls`. У нашего класса `AppComponent`, таким образом, есть одно свойство с именем `title`.

Использование свойства `templateUrl` для загрузки отдельного файла, содержащего наш HTML-шаблон, является примером принципа инверсии зависимостей. Класс `AppComponent` зависит от HTML-шаблона для визуализации компонента в браузере. При использовании свойства `template` у нас есть тесная связь между HTML-шаблоном и самим классом. Это означает, что любая модификация шаблона требует перекомпиляции модуля. Разделив шаблон на отдельный загружаемый файл, мы нарушили эту связь, и класс модуля можно изменить независимо от его HTML-шаблона.

Файл `app.component.html` сейчас выглядит очень просто:

```
<div>
   {{title}}
</div>

<div>
   <button>detail</button>
</div>
```

Здесь у нас есть два элемента `<div>`. Первый содержит наш заголовок, а второй – кнопку.

Использование Bootstrap и Font-Awesome

Теперь, когда у нас есть основы нашего приложения Angular, мы можем конкретизировать HTML-код, который он будет содержать. Для этого мы будем использовать фреймворк Bootstrap и иконки из Font-Awesome. Bootstrap – это основанный на HTML CSS-метод создания общих веб-компонентов, которые обеспечивают большую часть функциональности и стиля, необходимых для современных веб-сайтов. Bootstrap предоставляет простой синтаксис для добавления профессионально выглядящих стилей на наш сайт, от кнопок до значков, вкладок или предупреждений, а также практически всего, что находится между ними. Он был создан как адаптивный фреймворк, что означает, что он будет автоматически настраиваться для оптимальной визуализации на планшетах, мобильных или настольных устройствах.

Font-awesome – это набор иконок с открытым исходным кодом, который можно бесплатно использовать в коммерческих проектах, с очень широким диапазоном доступных иконок. Чтобы включить стили Bootstrap и Font-awesome в нашу веб-страницу, сначала нужно установить их через `npm`:

```
npm install bootstrap --save
npm install font-awesome -save
```

Чтобы включить файлы boostrap.css и font-awesome.css в наше приложение, мы можем просто отредактировать файл angular.json в базовом каталоге и добавить две записи в свойство styles:

```
"styles": [
   "styles.css",
   "./node_modules/bootstrap/dist/css/bootstrap.min.css",
   "./node_modules/font-awesome/css/font-awesome.css"
],
```

Теперь мы можем приступить к детализации дизайна на нашей странице app.component.html, начиная с панели навигации в верхней части экрана:

```
<nav class="navbar navbar-expand-lg navbar-dark bg-dark">
   <a class="navbar-brand"> </a>
   <ul class="navbar-nav mr-auto">
      <li class="nav-item nav-link active">Home</li>
      <li class="nav-item nav-link">Products</li>
      <li class="nav-item nav-link">About Us</li>
   </ul>
</nav>
```

Здесь мы создали верхнюю панель навигации, указав ссылку <nav>, и установили CSS-классы Bootstrap для создания панели навигации темного цвета, которая занимает полосу в верхней части страницы. Внутри ссылки <nav> у нас есть тег <a>, который является просто пустым элементом, а затем мы определяем дочерний элемент <ul> с тремя ссылками <li> внутри него. Эти ссылки называются Home, Products и About Us и будут отображаться как навигационные ссылки.

Обратите внимание, что для примеров в этой главе используется Bootstrap версии 4.1.3, которая имеет некоторые отличия от более ранних версий. Если предыдущая панель навигации отображается неправильно, проверьте файл package.json и убедитесь, что версия Bootstrap верна:

```
"dependencies": {
   .. другие npm-библиотеки ...
   "bootstrap": "^4.1.3",
   ... другие npm-библиотеки ...
}
```

Создание боковой панели

Теперь мы можем взглянуть на создание нашей левой боковой панели. Отличным ресурсом для HTML-элементов, CSS-стилей и анимации является сайт W3Schools (https://www.W3schools.com). В разделе с практическими рекомендациями представлена огромная библиотека примеров, включая слайд-шоу, модальные

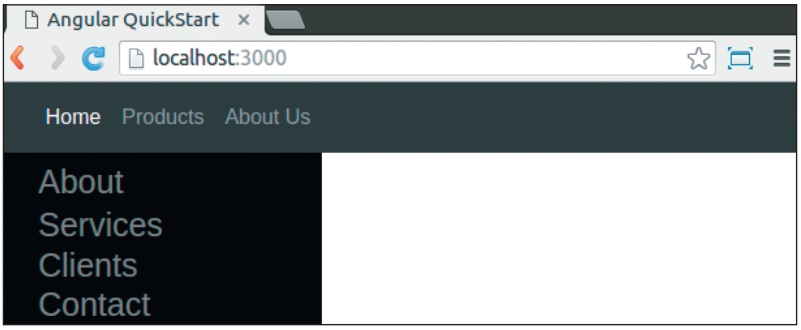
окна, индикаторы выполнения и адаптивные таблицы, и это лишь некоторые из них. Мы будем использовать пример из бокового раздела навигации, который называется `Sidenav Push Content`. В этом примере показано, как создать боковой экран навигации, который перемещает основное содержимое страницы по мере ее расширения, вместо того чтобы создавать наложение. Начнем с добавления HTML-кода в файл `app.component.html`:

```
<div id="mySidenav" class="sidenav">
   <a href="#">About</a>
   <a href="#">Services</a>
   <a href="#">Clients</a>
   <a href="#">Contact</a>
</div>
```

Здесь мы описали `<div>`-элемент с идентификатором `mySideNav` и CSS-класс `sidenav`. Этот элемент содержит четыре вложенные ссылки. Чтобы превратить это в привлекательную боковую панель навигации, нам теперь нужно отредактировать файл `app.component.css`, добавив несколько стилей:

```
/* Боковое меню навигации */
.sidenav {
   height: 100%; /* На всю высоту - 100% */
   width: 250px; /* Нулевая ширина - Изменяем это
                 с помощью JavaScript */
   position: fixed; /* Остается на месте */
   z-index: 1;   /* Помещается сверху */
   top: 55px;
   left: 0;
   background-color: #111; /* Цвет фона - чёрный */
   overflow-x: hidden; /* Отключаем горизонтальную прокрутку */
   padding-top: 60px; /* Размещаем содержимое на расстоянии 60px
                         от верхнего края */
   transition: 0.3s; }

/* Ссылки меню навигации */
.sidenav a {
   padding: 8px 8px 8px 32px;
   text-decoration: none;
   font-size: 25px;
   color: #818181;
   display: block;
   transition: 0.3s }

/* Когда вы наводите курсор мыши на ссылки меню, их цвет меняется */
.sidenav a:hover, .offcanvas a:focus{
   color: #f1f1f1;
}
```

После этого наша боковая панель навигации начинает обретать форму, как видно на скриншоте ниже:



Здесь у нас есть красиво оформленная боковая панель навигации. К сожалению, содержимое нашей главной страницы исчезло за этой боковой панелью навигации, а это означает, что нам потребуется применить окружающий -элемент и стили, чтобы левая панель перетаскивала содержимое основной панели вправо. Тогда содержимое нашей основной панели будет выглядеть так:

```
<div id="main" class="main-content-panel">
   <div class="row">
      <div class="col-sm-1">
         <button>
            <span class="fa fa-chevron-left"> </span>
         </button>
      </div>
      <div class="col-sm-11">
         <div class="row-content-header">{{title}}</div>
      </div>
   </div>
   <div class="main-content">
      <button class="btn btn-primary">
         detail
      </button>
   </div>
</div>
```

Здесь мы обернули основное содержимое в элемент `<div>` с идентификатором `"main"` и классом `"main-content-panel"`. Теперь этот элемент разбивается на ряд, состоящий из двух колонок, размерами 1 и 11. В этом ряду находится наша кнопка показа/сокрытия боковой панели и элемент `{{title}}`. Под этим рядом заголовка находится наше основное содержимое, которое включает в себя одну кнопку с надписью `detail`. Наш соответствующий CSS-файл для этого раздела HTML-кода выглядит так:

```
#main {
  margin-left: 250px;
  transition: .3s;
}

#main-body {
  transition: .3s;
}

.main-content {
  padding: 20px;
}

.row-content-header {
  padding: 5px;
  font-size: 20px;
}
```

Здесь есть два ключевых стиля, которые влияют на содержимое нашей страницы. Первый – элемент `margin-left: 250px` стиля `#main`. Значение `margin-left` является CSS-свойством, которое сдвигает наше основное содержимое вправо, когда видна левая панель. Это свойство соответствует значению соответствующей боковой панели `.sidenav {width: 250px;}`. Другими словами, ширина боковой панели составляет `250` пикселей, а основная панель имеет отступ слева `250` пикселей. Эти два значения вместе показывают левую панель, а также сдвигают основную панель вправо. Мы изменим эти значения с `250` на `0` пикселей, чтобы показать или скрыть левую панель.

Второй ключевой стиль – это свойство `transition: .3s;`, определяющее, сколько времени потребуется для анимации сворачивания и развертывания боковой панели, а также сдвига основной панели вправо или ее расширения для заполнения экрана. Имея эти стили, мы можем теперь прикрепить некий код, чтобы начать анимированный переход. Чтобы это работало, нам нужно зарегистрировать обработчик кликов в HTML-коде, а затем реализовать его в файле `app.component.ts`.

Сначала рассмотрим DOM-событие `button click` в файле `app.component.html`:

```
<button (click)="showHideSideClicked()">
  <span class="fa fa-chevron-left"> </span>
</button>
```

Здесь мы определили функцию `showHideSideClicked`, которая будет вызываться всякий раз, когда мы нажимаем кнопку **Показать/Скрыть**. Наши изменения в файле `app.component.ts` будут такими:

```
export class AppComponent
{
   title = "Select an option :";
   isSideNavVisible = true;
   showHideSideClicked() {
      if (this.isSideNavVisible) {
         document.getElementById('main')
            .style.marginLeft = "0px";
         document.getElementById('mySidenav')
            .style.width = "0px";
         this.isSideNavVisible = false;
      } else {
         document.getElementById('main')
            .style.marginLeft = "250px";
         document.getElementById('mySidenav')
            .style.width = "250px";
         this.isSideNavVisible = true;
      }
   }
}
```

Здесь мы добавили в класс AppComponent свойство `isSideNavVisible` и установили для него значение `true` по умолчанию. Это свойство отслеживает, является боковая панель навигации видимой или нет. Затем мы реализовали функцию `showHideSideClicked`. Если боковая панель навигации видима, мы устанавливаем стиль главной панели `marginLeft` в `0px`, а также устанавливаем ширину элемента `mySideNav` в `0px`. Это, по существу, приводит к сворачиванию боковой панели и заставляет основную панель заполнять весь экран. Если боковая панель навигации свернута, мы делаем противоположное и также устанавливаем в то же самое время свойство `isSideNavVisible`. При запуске нашего приложения на этом этапе левая панель довольно хорошо показывается и скрывается, используя свойство `transition: .3s` для применения визуально привлекательной анимации.

Создание наложения

Теперь мы можем обратить наше внимание на вторую страницу, которая будет перемещаться влево, когда мы нажмем кнопку `detail`. Наш фрагмент HTML-кода выглядит так:

```
<div id="mySidenav" class="sidenav">
...имеющаяся боковая панель...
</div>

<div id="myRightScreen" class="overlay">
   <button class="btn button-no-borders"
      (click)="closeClicked()">
```

```
        <span class="fa fa-chevron-left"></span>
    </button>
    <div class="overlay-content">
      <h1>page 2</h1>
    </div>
</div>
...имеющаяся основная панель...
<div id="main" class="main-content-panel">
```

Здесь мы вставили -элемент с идентификатором myRightScreen и указали CSS-класс overlay. Это простой элемент, который содержит кнопку в верхней части, обработчик щелчка closeClicked и элемент <h1> для отображения второй страницы. Как и в случае с боковой панелью навигации, нам понадобятся CSS-стили для выполнения двух вещей. Во-первых, нам нужно переместить вторую страницу вправо, а затем как-то сдвинуть ее влево, когда нажимается кнопка detail. Наш CSS-код выглядит так:

```
/* Наложение (фон) */
.overlay {
  height: 100%;
  width: 100%;
  position: fixed; /* Остается на месте */
  z-index: 1; /* Находится сверху */
  left: 0;
  top: 54px;
  overflow-x: hidden; /* Отключаем горизонтальную прокрутку */
  transition: 0.3s;
  transform: translateX(100%);
  border-left: 1px solid;
}
```

Здесь есть два стиля, которые управляют тем, как открывается вторая страница. Первый – это стиль transform: translateX(100%), а второй – это стиль transition: 0.3s. Стиль преобразования в этом случае, по существу, перемещает начальную позицию X -элемента на 100%. Это означает, что по умолчанию он смещен по оси X на 100% ширины страницы и поэтому не виден. Стиль transition: 0.3s снова просто анимирует показ или скрытие панели.

Давайте реализуем ряд обработчиков кликов на нашей странице, чтобы увидеть это в действии. Во-первых, нам нужно обработать событие click кнопки detail:

```
buttonClickedDetail() {
  document.getElementById('myRightScreen')
    .style.transform = "translateX(0%)";
  document.getElementById('main')
    .style.transform = "translateX(-100%)";
}
```

Здесь мы делаем две вещи. Во-первых, мы устанавливаем для свойства второй страницы transform значение `translateX(0%)` в противоположность `translateX(100%)`, устанавливая начальную позицию X `<div>`-элемента на `0%`. Имея CSS-свойство `translate`, мы получаем нужный нам эффект скольжения влево. Второе, что мы делаем в этой функции, – это устанавливаем для свойства нашего основного `<div>`-элемента `transform` значение `translateX(-100%)`. Опять же, это дает эффект сдвига главной панели вправо. Прежде чем мы протестируем этот переход, давайте реализуем функцию `closeClicked`, которая закрывает правую панель:

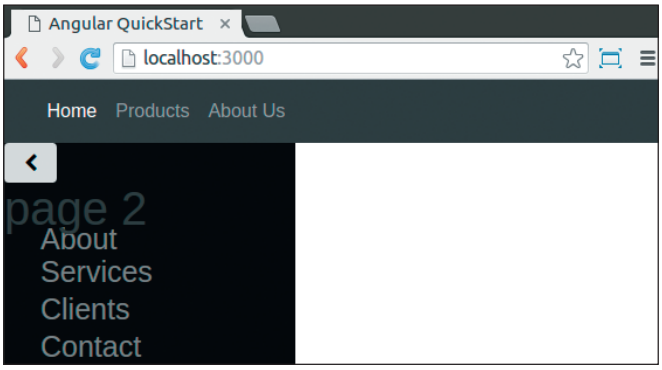
```
closeClicked() {
   document.getElementById('myRightScreen')
      .style.transform = "translateX(100%)";
   document.getElementById('main')
      .style.transform = "translateX(0%)";
}
```

Здесь мы выполняем действие, противоположное функции `buttonClicked-Detail`, чтобы сдвинуть панель на второй странице вправо, а также открыть нашу главную панель. Эти две функции работают совместно для установки свойств `translateX` `<div>`-элемента `myRightScreen` и основного `<div>`-элемента.

Если мы запустим сейчас нашу страницу, то сможем нажать на кнопку **Detail** и увидеть, что вторая страница сдвигается влево.

Координация переходов

Пока мы создали простое веб-приложение с основной панелью, левой боковой панелью и панелью второй страницы, а также добавили несколько CSS-стилей и переходов для создания визуально приятной структуры страницы. К сожалению, у нашей текущей реализации есть ряд проблем. Если мы находимся на главной странице и наша левая боковая панель видна, то при нажатии на кнопку **Detail** левая панель не закрывается перед сдвигом правой панели. Это приводит к отображению второй страницы в верхней части левой панели:



Здесь видно, что вторая страница, которая сдвинулась влево, отображается в верхней части панели навигации. Чтобы исправить это, мы могли бы вызвать функцию `showHideSideClicked`, которая у нас уже есть, чтобы сначала скрыть боковую панель. Кажется, это решает проблему, но приводит к появлению еще одной ошибки. Если мы видим боковую панель, а затем показываем и скрываем панель `Detail`, боковая панель остается закрытой. Чтобы исправить эту ошибку, мы могли бы снова вызвать функцию `showHideSideClicked`, когда закрываем правую панель, но это решение, к сожалению, имеет свои странные ошибки.

Хотя мы могли бы переработать логику нашего приложения, чтобы устранить все эти ошибки, мы быстро впадаем в обескураживающий цикл попыток исправить одну вещь, только чтобы обнаружить, что она имеет другой нежелательный побочный эффект. Что нам действительно нужно, так это механизм для отслеживания всех этих визуальных элементов и контроля того, как приложение реагирует на пользовательский ввод. Здесь на помощь приходят шаблоны проектирования **State** и **Mediator**.

Шаблон State

«Банда четырех» описывает два шаблона проектирования, State и Mediator. Шаблон State использует набор конкретных классов, унаследованных от базового класса, для описания конкретного состояния. В качестве примера рассмотрим создание перечисления для описания состояний двери. На первый взгляд, дверь может быть либо открытой, либо закрытой. В этом случае простой поток управления `if ... else`, скорее всего, позаботится о любой логике, которую мы хотим применить.

Рассмотрим, однако, что происходит с нашим потоком управления и логикой, если нам нужно состояние `Locked` и `Unlocked` или если это раздвижная дверь с состояниями `SlightlyAjar`, `HalfOpen`, `AlmostFullyOpen` и `FullyOpen`. Шаблон проектирования State позволяет нам легко определять эти состояния и корректировать нашу логику на основе текущего состояния объекта.

Если немного подумать о нашем приложении, мы знаем, что наши экраны будут находиться в том или ином конкретном состоянии в любой момент времени. Мы либо на панели основного экрана, либо на панели второй страницы. Кроме того, левая боковая панель либо видна, либо скрыта. Эта комбинация дает нам три состояния:

- только основная панель;
- основная панель с боковой навигацией;
- панель с подробностями.

Интерфейс шаблона State

Шаблон State помогает нам определять эти состояния в коде. Основной принцип шаблона проектирования State заключается в том, что мы создаем интерфейс или абстрактный базовый класс, который определяет свойства каждого состояния, а затем мы создаем конкретные классы для каждой специализации. Таким образом, в нашем приложении у нас есть два основных вопроса, которые мы должны задать каждому состоянию:

- видна ли боковая панель?
- мы на главной панели или на панели с подробностями?

Кроме того, если мы находимся на главной панели, то нам также нужно знать, показывать в левом верхнем углу главной панели стрелку > или стрелку <. Это связано с тем, видна боковая панель или нет. Давайте создадим файл с именем `StateMediator.ts` для хранения наших интерфейсов:

```
export enum StateType {
   MainPanelOnly,
   MainPanelWithSideNav,
   DetailPanel
}

export enum PanelType {
   Primary,
   Detail
}

export interface IState {
   getPanelType(): PanelType;
   getStateType(): StateType;
   isSideNavVisible(): boolean;
   getPanelButtonClass(): string;
}
```

Мы начинаем с перечисления `StateType`, которое позволяет нам узнать, в каком из трех состояний мы находимся. Затем мы определяем перечисление `PanelType` для того, находимся ли мы на панелях `Primary` или `Detail`. У нашего интерфейса `IState` есть четыре функции. `getPanelType` возвращает значение перечисления `PanelType`, а функция `getStateType` возвращает значение перечисления `StateType`. Функция `isSideNavVisible` просто возвращает значение `boolean`, указывающее, видна боковая панель навигации или нет. Последняя функция, `getPanelButtonClass`, возвращает имя класса для переключения кнопки **Показать/Скрыть** со значка < на значок >, в зависимости от состояния боковой панели.

После этого мы определили, какие вопросы можем задать каждому из наших конк ретных классов состояния. В зависимости от того, находимся ли мы на

главной панели или на панели с подробностями, ответ на этот вопрос будет немного меняться. В этом суть шаблона проектирования State. Определите интерфейс, который даст вам ответы, необходимые для всех состояний, а затем запрограммируйте этот интерфейс. Это защищает любую логику, которую мы строим, чтобы потреблять эти состояния из определения самих состояний. Другими словами, добавление или удаление нового класса состояний не повлияет на код, который мы написали для интерфейса `IState`.

Конкретные состояния

Теперь давайте рассмотрим три конкретных класса состояний:

```
export class MainPanelOnly

implements IState {
   getPanelType(): PanelType {return PanelType.Primary;}
   getStateType(): StateType {return StateType.MainPanelOnly;}
   getPanelButtonClass(): string {return 'fa-chevron-right';}
   isSideNavVisible(): boolean {return false;}
}
```

Мы начнем с класса состояний `MainPanelOnly`, который используется для описания состояния, когда боковая панель навигации не видна, и мы находимся на главной панели просмотра. Это очень простой класс, который реализует интерфейс `IState` и поэтому просто возвращает правильные значения для каждой из четырех функций. Как видно из возвращаемых значений, мы в `PanelType.Primary`, функция `isSideNavVisible` возвращает `false`, и нам нужен класс `'fa-chevron-right'` для отображения стрелки на нашей кнопке **Показать/Скрыть**. Наши два других конкретных состояния очень похожи:

```
export class MainPanelWithSideNav
   implements IState {
      getPanelType(): PanelType {return PanelType.Primary;}
      getStateType(): StateType {return StateType.
         ainPanelWithSideNav;}
      getPanelButtonClass(): string {return 'fa-chevron-left';}
      isSideNavVisible(): boolean {return true;}
   }

export class DetailPanel
   implements IState {
      getPanelType(): PanelType {return PanelType.Detail;}
      getStateType(): StateType {return StateType.DetailPanel;}
      getPanelButtonClass() : string {return '';}
      isSideNavVisible(): boolean {return false;}
   }
```

Здесь класс состояния `MainPaleWithSideNav` такой же, как и класс `MainPanel`, за исключением того, что он возвращает значение `true` для функции `isSideNavVisible` и `'fa-chevron-left'` для класса `PanelButton`. Класс состояния `DetailPanel` возвращает `PanelType.Detail`, значение `false` для функции `isSideNavVisible` и пустое имя класса для кнопки панели.

Эти три класса очень просты и описывают состояние, в котором должны быть элементы пользовательского интерфейса, когда они находятся в текущем состоянии. Эти классы помогают нам инкапсулировать логику, которая используется в нашем приложении для управления различными элементами пользовательского интерфейса на нашем экране.

Шаблон Mediator

Теперь, когда мы можем описать различные состояния, в которых находится наш пользовательский интерфейс, мы можем начать применять логику, необходимую для перемещения между этими состояниями. Мы будем использовать шаблон Mediator (Посредник) для достижения этой цели. Цель данного шаблона состоит в том, чтобы определить, как набор объектов взаимодействует друг с другом, и делает это путем внедрения объекта между объектами, которые влияют друг на друга. Это означает, что рассматриваемые объекты на самом деле не взаимодействуют друг с другом, они работают против интерфейса. Это способствует слабой связанности между объектами.

Существует две части шаблона Mediator. Первая часть – определить интерфейс, который может вызывать посредник, чтобы применить необходимые изменения. Посредник в данном случае общается с нашими классами пользовательского интерфейса. В нашем приложении нам потребуется посредник, чтобы иметь возможность дать интерфейсу пользователя сигнал, дабы показать или скрыть боковую панель навигации и показать или скрыть панель подробностей. Посреднику также необходимо переключить кнопку **Показать/Скрыть** с < на > в зависимости от текущего состояния приложения.

Вторая часть шаблона – это логика перехода из одного состояния в другое. Посредник сам будет отслеживать, в каком состоянии находится приложение, а затем координировать действия пользовательского интерфейса для перехода из одного состояния в другое.

Определяя интерфейс для этих взаимодействий, мы следуем передовым методикам объектно-ориентированного программирования и защищаем код посредника от фактической реализации меняющейся логики пользовательского интерфейса. Мы также можем кодировать и тестировать логику посредника без фактического пользовательского интерфейса.

Поэтому наш интерфейс для класса посредника выглядит так:

```
export interface IMediatorImpl {
   showNavPanel();
   hideNavPanel();
   showDetailPanel();
   hideDetailPanel();
   changeShowHideSideButton(fromClass: string, toClass: string);
}
```

Здесь мы разделили все изменения пользовательского интерфейса, необходимые нашему приложению, на пять функций. Мы можем либо показать, либо скрыть боковую панель навигации, показать или скрыть панель подробностей или обновить кнопку CSS-класса.

Оглядываясь на проделанную работу, мы упростили нашу бизнес-логику, разделив ее на две части. Во-первых, мы определили состояния, в которых будет находиться наш пользовательский интерфейс в любой момент времени, а во-вторых, определили функции, необходимые для обновления нашего пользовательского интерфейса. Мы уже на пути к созданию модульного, объектно-ориентированного, простого для понимания и простого в обслуживании приложения. Мы рассмотрим реализацию логики посредника чуть позже, после того как выполним ряд действий с нашим существующим кодом.

Модульный код

Пока что у нашего приложения есть HTML- и CSS-код и бизнес-логика как часть класса `AppComponent`. Хотя мы уже разбили этот класс на отдельные файлы `app.component.html` и `app.component.css`, в действительности он содержит несколько отдельных компонентов в одном. Давайте воспользуемся этой возможностью, чтобы модуляризировать наш код и создать три отдельных класса:

- класс `NavbarComponent` для визуализации и обработки панели навигации в верхней части экрана;
- класс `SideNavComponent` для визуализации левой боковой панели навигации;
- `RightScreenComponent` для обработки панели подробностей, которая сдвигается влево.

Это означает, что класс `AppComponent` становится центральным классом приложения и будет отвечать за координацию каждого из этих компонентов.

Компонент Navbar

Наша первая задача – создать класс `NavbarComponent`, который будет нести ответственность за визуализацию панели навигации в верхней части экрана. Для

этого мы создадим файлы `navbar.component.ts` и `navbar.component.html` в каталоге нашего приложения.

Это легко сделать с помощью интерфейса командной строки Angular, выполнив в командной строке:

```
ng generate component navbar
```

С помощью этой команды будет создан новый каталог в папке `src/app` и сгенерированы необходимые для нашего нового компонента файлы, включая основной файл `.ts`, файлы `.css`, `.html`, а также `.spec.ts` для тестирования. Еще будет сгенерирован стандартный код для создания компонента и изменения файла `app.module.ts`, чтобы автоматически сделать этот компонент доступным для использования.

Содержимое файла `navbar.component.html` можно просто скопировать из существующего файла `app.component.html`:

```
<nav class="navbar navbar-inverse bg-inverse
navbar-toggleable-sm">
   <a class="navbar-brand"> </a>
   <div class="nav navbar-nav">
      <a class="nav-item nav-link active">Home</a>
      <a class="nav-item nav-link">Products</a>
      <a class="nav-item nav-link">About Us</a>
   </div>
</nav>
```

Интерфейс командной строки Angular создаст для нас класс `NavbarComponent`:

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

@Component({
   selector: 'app-navbar',
   templateUrl: './navbar.component.html',
   styleUrls: ['./navbar.component.css']
})

export class NavbarComponent implements OnInit {
   constructor() { }
   ngOnInit() { }
}
```

Это очень простой класс Angular, который обращается к нашему HTML-файлу и определяет свойство селектора в декораторе `@Component`. Чтобы использовать этот новый компонент в нашем приложении, мы просто добавим тег `<app-navbar>` в наш текущий файл `app.component.html`.

Обратите внимание, что интерфейс командной строки Angular изменил файл `app.module.ts`, чтобы зарегистрировать для нас компонент:

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

@NgModule({
   declarations: [
      AppComponent,
      NavbarComponent
   ],
   imports: [
      BrowserModule
   ],
   providers: [],
   bootstrap: [AppComponent]})
```

Обратите внимание на оператор импорта модуля, который обращается к нашему модулю `navbar.component`, и обновленный массив `declarations` декоратора `@NgModule`. При обновлении массива `declarations` тег `<app-navbar>` будет доступен для использования в любом HTML-шаблоне.

Компонент SideNav

Аналогичным образом давайте теперь создадим компонент для левой навигационной панели:

```
ng generate component sidenav
```

Будут созданы три исходных файла в каталоге `app/sidenav`, как мы видели ранее, и будет зарегистрирован наш компонент с помощью изменения файла `app.module.ts`. Чтобы создать файл `sidenav.component.htm`, мы снова просто скопировали и вставили содержимое из нашего существующего HTML-файла:

```
<div id="mySidenav" class="sidenav">
   <a href="#">About</a>
   <a href="#">Services</a>
   <a href="#">Clients</a>
   <a href="#">Contact</a>
</div>
```

Автоматически сгенерированный класс `SideNavComponent` необходимо обновить с помощью двух новых функций, а именно:

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

@Component( {
   selector: 'app-sidenav',
```

```
    templateUrl: './sidenav.component.html',
    styleUrls: ['./sidenav.component.css']
})

export class SideNavComponent {
    closeNav() {
        document.getElementById('mySidenav')
            .style.width = "0px";
    }
    showNav() {
        document.getElementById('mySidenav')
            .style.width = "250px";
    }
}
```

В этом классе мы создали две функции, `closeNav` и `showNav`. Они устанавливают CSS-свойство `style.width` в `0px` и `250px` соответственно. Что мы сделали здесь, так это, по сути, инкапсулировали всю функциональность, окружающую боковую панель навигации, в класс `SideNavComponent`. Это принцип единственной ответственности – он гласит, что класс должен нести единственную ответственность.

Нам также необходимо удалить css из файла `app.component.css`, который применяется к этой панели `sidenav`, и поместить его в файл `sidenav.component.css`. Это пример инкапсуляции. У компонента `sidenav` есть все необходимое, включая стили, HTML-шаблоны, тесты и код, инкапсулированный в единый элемент.

Теперь мы можем вставить тег `<app-sidenav>` в файл `app.component.html`.

Компонент RightScreen

Давайте теперь продолжим и создадим последний компонент в нашем приложении, который занимает правый экран или экран с подробной информацией, под названием `RightScreenComponent`. Опять же, он генерируется так:

```
ng generate component rightscreen
```

В результате этого будут созданы файлы `rightscreen.component.ts`, HTML-файл и CSS-файл для этого компонента. Мы пока не будем фокусироваться на HTML- и CSS-файлах данного компонента, так как они просто скопированы и вставлены из существующих файлов `app.component`. Давайте вместо этого посмотрим на обновления класса Angular этого компонента, а именно:

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

@Component( {
    selector: 'rightscreen-component',
```

```
    templateUrl: './rightscreen.component.html',
    styleUrls: ['./rightscreen.component.css']
})

export class RightScreenComponent {
    @Output() notify: EventEmitter<string>
        = new EventEmitter<string>();
    closeClicked() {
        this.notify.emit('Click from nested component');
    }

    closeRightWindow() {
        document.getElementById('myRightScreen')
            .style.transform = "translateX(100%)";
    }

    openRightWindow() {
        document.getElementById('myRightScreen')
            .style.transform = "translateX(0%)";
    }
}
```

Здесь мы определили три функции: `closeClicked`, `closeRightWindow` и `openRightWindow`. Функции `closeRightWindow` и `openRightWindow` устанавливают значения `style.transform` для этого компонента, как мы уже обсуждали ранее.

Действительно интересной частью этого класса является функция `closedClicked` и использование того, что носит название класса `EventEmitter`. Обратите внимание, что мы импортировали и класс `EventEmitter`, и декоратор `Output` в нашем операторе `import` в верхней части файла. Angular использует декоратор свойства `@Output` и обобщенный класс `EventEmitter`, чтобы дать возможность компонентам уведомить другие компоненты, когда происходят события.

Помните, что в верхней панели слева есть кнопка, которая используется для закрытия панели и возврата к главному экрану. HTML-код этой кнопки выглядит так:

```
<button class="btn btn-default" (click)="closeClicked()">
    <span class="fa fa-chevron-left"></span>
</button>
```

Мы знаем, что синтаксис Angular для обработки DOM-события `click` – это `(click) = "<handlerFunction>"`. В нашем предыдущем HTML-коде `<handlerFunction>` называется `closeClicked` и поэтому должно быть определено в самом классе нашего компонента.

Однако класс `RightScreenComponent` контролирует только свою собственную HTML-область и поэтому не может включать в себя какие-либо функции, пред-

назначенные для самого приложения. Тогда в этом случае все, что нам нужно сделать, – это вызвать событие, сообщающее, что была нажата кнопка **Закрыть**, и предоставить другой части приложения реагировать на это событие и делать что-либо. Опять же, это связано с принципом единственной ответственности.

Давайте подробнее рассмотрим синтаксис `EventEmitter`:

```
@Output() notify: EventEmitter<string>
   = new EventEmitter<string>();

closeClicked() {
   this.notify.emit('Click from nested component');
}
```

Мы настроили наш источник событий с помощью декорации свойства `notify`, используя декоратор `@Output`. Затем мы указываем, что тип этого свойства – `EventEmitter<string>`, а после немедленно создаем экземпляр класса `EventEmitter`. Класс `EventEmitter` – обобщенный класс, что означает, что мы можем заменить значение `<number>` или `<boolean>` или даже сложный класс в этом объявлении.

Поскольку свойство `notify` является экземпляром класса `EventEmitter` типа `string`, мы можем вызвать `this.notify.emit` со строковым аргументом в нашей функции `closeClicked`. Это обеспечивает передачу события, когда пользователь нажимает эту кнопку в классе `RightScreenComponent`.

Теперь нам нужно определить обработчик события для этого события. Поскольку класс `AppComponent` отвечает за создание и управление классом `RightScreenComponent`, мы вносим изменения в наш файл `app.component.html`, чтобы зарегистрироваться для этого события уведомления. Включение тега `<app-rightscreen>` теперь выглядит так:

```
<app-rightscreen (notify)='onNotifyRightWindow($event)'>
</app-rightscreen>
```

Здесь мы добавили атрибут в наш тег `<app-rightscreen>`, чтобы зарегистрировать событие (`notify`), а затем вызвали функцию `onNotifyRightWindow` в классе `AppComponent`. Реализация этой функции на данный момент может просто вызвать предупреждение, чтобы мы могли проверить, работает ли правильно запуск и регистрация этого события:

```
onNotifyRightWindow(message:string):void {
   alert('clicked');
}
```

Мы подключим этот обработчик событий чуть позже к нашему классу посредника, чтобы перевести переключатель в другое состояние.

Дочерние компоненты

Класс AppComponent является владельцем всего нашего приложения. Он визуализирует HTML-код, используемый для всей страницы, navbar, sidenav, Rightscreen и компоненты главной панели. Как таковой он также является родителем этих подкомпонентов. Другими словами, все эти компоненты являются дочерними элементами класса AppComponent и называются дочерними компонентами. Теперь нам нужно найти способ, чтобы класс AppComponent мог обращаться к классам SideNavComponent и RightScreenComponent внутри самого себя. Это нужно, чтобы связать экземпляры этих классов, созданные с помощью HTML-тегов, <app-sidenav> и <app-rightscreen>.

Для этого Angular предоставляет декоратор свойств @ViewChild. Чтобы использовать его, наш класс AppComponent необходимо обновить:

```
import { Component, ViewChild } from '@angular/core';
import { SideNavComponent } from './sidenav.component';
import { RightScreenComponent } from './rightscreen.component'
.. @Component ...
export class AppComponent {
   @ViewChild(SideNavComponent)
   private sideNav : SideNavComponent;
   @ViewChild(RightScreenComponent)
   private rightScreen: RightScreenComponent;
.. остальная часть класса ...
```

Нам нужно сделать несколько изменений. Во-первых, нам необходимо импортировать декоратор ViewChild из модуля @angular/core, а затем импортировать модули SidenavComponent и RightScreenComponent. Во-вторых, нам нужно создать два закрытых свойства, sideNav и rightScreen, чтобы хранить экземпляры наших дочерних компонентов.

Затем мы используем декоратор @ViewChild от Angular с именем класса, к которому хотим обратиться. Это означает, что функция @ViewChild(SidenavComponent) подключит частное свойство sideNav к правильному экземпляру класса SidenavComponent.

Точно так же мы попросим Angular подключить экземпляр класса RightscreenComponent, используемого в нашем HTML-коде, к закрытой переменной rightScreen. Таким образом, наш класс AppComponent теперь имеет программный доступ к этим двум классам, к которым обратились в HTML-коде.

Реализация интерфейса посредника

Теперь, когда у класса AppComponent есть ссылки на его дочерние компоненты, мы можем сосредоточиться на реализации интерфейса IMediatorImpl в этом классе:

```
export class AppComponent implements IMediatorImpl {

   ...существующий код...

   showNavPanel() {
      this.sideNav.showNav();
      document.getElementById('main').style.marginLeft = "250px";
   }
   hideNavPanel() {
      this.sideNav.closeNav();
      document.getElementById('main').style.marginLeft = "0px";
   }
   showDetailPanel() {
      this.rightScreen.openRightWindow();
      document.getElementById('main').style.transform =
         "translateX(-100%)";
   }
   hideDetailPanel() {
      this.rightScreen.closeRightWindow();
      document.getElementById('main').style.transform =
         "translateX(0%)";
   }
   changeShowHideSideButton(fromClass: string, toClass: string) {
      if (fromClass.length > 0 && toClass.length > 0) {
         document.getElementById('show-hide-side-button')
            .classList.remove(fromClass);
         document.getElementById('show-hide-side-button')
            .classList.add(toClass);
      }
   }
}
```

Вначале идет функция `showNavPanel`, которая вызывает реализацию функции `showNav` для дочернего компонента `sideNav`, а затем устанавливает стиль `marginLeft` для основного элемента DOM. Аналогично, функция `hideNavPanel` делает обратное. Функция `showDetailPanel` вызывает реализацию функции `openRightWindow` для дочернего компонента `rightScreen`, а затем устанавливает свойство `transform` для главного элемента DOM.

Теперь мы можем сосредоточиться на самом классе `Mediator`.

Класс Mediator

Класс `Mediator` отвечает за координацию общего состояния приложения и взаимодействия между различными классами пользовательского интерфейса. Как таковой он действительно должен иметь три ключевые составляющие. Во-пер-

вых, это конкретная реализация интерфейса `IMediatorImpl`, чтобы он мог вызывать различные функции, необходимые для обновления пользовательского интерфейса. Мы только что реализовали интерфейс `IMediatorImpl` в классе `AppComponent`, поэтому нам нужно будет передать ссылку на экземпляр `AppComponent` в класс `Mediator`.

Во-вторых, классу `Mediator` нужен конкретный экземпляр каждого из наших классов состояния, чтобы он мог воссоздать как текущее состояние приложения, так и желаемое последующее состояние. Затем он может сравнить текущее и желаемое последующее состояния, чтобы выяснить, какие изменения должны произойти, чтобы переходить из одного состояния в другое.

В-третьих, класс `Mediator` должен хранить текущее состояние приложения. Поскольку он отвечает за переход из одного состояния в другое, для посредника имеет смысл быть единственным источником истины для всего, что связано с состояниями. Кроме того, если функциональность пользовательского интерфейса зависит от текущего состояния приложения, мы можем направлять любые запросы о том, что делать через класс `Mediator`, чтобы он принимал решение за нас.

Учитывая все это, давайте взглянем на свойства и конструктор нашего класса `Mediator`:

```
export class Mediator {
private _mainPanelState = new MainPanelOnly();
private _detailPanelState = new DetailPanel();
private _sideNavState = new MainPanelWithSideNav();

private _currentState: IState;
private _currentMainPanelState: IState;
private _mediatorImpl: IMediatorImpl;

constructor(mediatorImpl: IMediatorImpl) {
   this._mediatorImpl = mediatorImpl;
   this._currentState = this._currentMainPanelState =
      this._sideNavState;
}
```

Мы начнем с трех конкретных экземпляров трех наших классов состояния с именами `_mainPanelState`, `_detailPanelState` и `_sideNavState`. Далее у нас идут два свойства, `_currentState` и `_currentMainPanelState`, оба из которых имеют тип `IState`. Эти свойства будут использоваться для хранения текущего состояния самого приложения и основной панели. Помните, что если мы переключимся с главной панели на панель подробностей, а затем вернемся назад, боковая панель навигации должна снова появиться в том же состоянии, в котором мы ее оставили. Вот для чего будет использоваться переменная состояния `_currentMainPanel`.

Следующая функция, которую мы реализуем в классе `Mediator`, – простая фабричная функция для извлечения конкретного экземпляра объекта состояния с учетом значения перечисления `StateType` в качестве входных данных:

```
getStateImpl(stateType: StateType) : IState {
   var stateImpl : IState;
   switch(stateType) {
      case StateType.DetailPanel:
         stateImpl = this._detailPanelState;
         break;
      case StateType.MainPanelOnly:
         tateImpl = this._mainPanelState;
         break;
      case StateType.MainPanelWithSideNav:
         stateImpl = this._sideNavState;
         break;
   }
   return stateImpl;
}
```

Это простая вспомогательная функция, которая возвращает правильную реализацию объекта состояния с учетом значения перечисления `StateType`.

Теперь мы можем сосредоточиться на сердце класса `Mediator` – управлении изменениями пользовательского интерфейса при переходе из одного состояния в другое:

```
moveToState(stateType: StateType) {
   var previousState = this._currentState;
   var nextState = this.getStateImpl(stateType);

   if (previousState.getPanelType() == PanelType.Primary &&
      nextState.getPanelType() == PanelType.Detail ) {
         this._mediatorImpl.showDetailPanel();
   }
   if (previousState.getPanelType() == PanelType.Detail &&
      nextState.getPanelType() == PanelType.Primary) {
      this._mediatorImpl.hideDetailPanel();
   }

   if (nextState.isSideNavVisible())
      this._mediatorImpl.showNavPanel();
   else
      this._mediatorImpl.hideNavPanel();

   this._mediatorImpl.changeShowHideSideButton(
      previousState.getPanelButtonClass(),
```

```
        nextState.getPanelButtonClass() );

    this._currentState = nextState;
    if (this._currentState.getPanelType() == PanelType.Primary ) {
      this._currentMainPanelState = this._currentState;
    }
  }
```

Функция `moveToState` содержит всю логику пользовательского интерфейса для обработки трех состояний нашего приложения. Мы начнем с объявления двух переменных `previousState` и `nextState`. Переменная `previousState` находится там, где сейчас находимся мы, а переменная `nextState` – там, где мы хотим быть, как передано через аргумент `stateType`.

Имея эти два объекта состояния, мы можем начать сравнивать их свойства, а затем соответствующим образом вызвать функции интерфейса `IMediatorImpl`.

Рассмотрим первый оператор `if`. Логика здесь просто утверждает следующее:

- если мы были на основной панели и хотим перейти на панель подробностей, просим пользовательский интерфейс показать панель подробностей.

Второй оператор `if` гласит следующее:

- если мы находимся на панели подробностей и хотим перейти на основную панель, просим пользовательский интерфейс скрыть панель подробностей.

Наш третий оператор `if` гласит следующее:

- если наше состояние сообщает нам, что боковая панель навигации должна быть видимой, покажите ее, в противном случае скройте ее.

Затем мы вызываем пользовательский интерфейс, чтобы переключить кнопку **Показать**/**Скрыть** с нашего значка текущего состояния на значок будущего состояния. Это приведет к переключению кнопки с < на > или наоборот, в зависимости от свойств двух наших состояний.

После того как мы закончили обновление пользовательского интерфейса, нам нужно сохранить наше текущее состояние.

Наконец, наш последний оператор `if` утверждает, что если мы находимся на главной панели, обновите внутреннее значение свойства `_currentMainPanelState`. Нам нужно сохранить это значение, чтобы при переключении на панель подробностей и обратно мы правильно восстанавливали боковую панель навигации.

Эта функция в простых, понятных человеку выражениях содержит информацию о том, как переходить из одного состояния в другое. Наша логика сводилась к тому, чтобы задать несколько простых вопросов и ответить соответственно.

Использование Mediator

Последний шаг в реализации шаблонов проектирования State и Mediator – инициирование изменения состояния. Инициатор может находиться исключительно в нашем коде или быть результатом действий с нашим пользовательским интерфейсом. Для начала нам нужно будет создать новый экземпляр класса Mediator и зарегистрировать наш класс AppComponent в качестве реализации для интерфейса IMediatorImpl:

```
export class AppComponent
   implements IMediatorImpl
{
   ... существующий код ...
   mediator: Mediator = new Mediator(this);
```

Здесь мы указываем, что класс AppComponent реализует интерфейс IMediatorImpl, а затем определяем локальную переменную mediator. Она вызывает конструктор Mediator, передавая this (экземпляр класса AppComponent). Этот вызов, по сути, регистрирует класс AppComponent в качестве реализации интерфейса IMediatorImpl, который Mediator использует для внесения изменений в пользовательский интерфейс.

После того как мы зарегистрировали класс AppComponent в Mediator, мы можем использовать Mediator для запуска изменения состояния. В качестве примера давайте убедимся, что при первом запуске приложения мы показываем только основную панель, или, другими словами, переходим в состояние StateType.MainPanelOnly. Для этого нам нужно подключиться к жизненному циклу рендеринга компонентов Angular и реализовать функцию ngAfterViewInit:

```
export class AppComponent
   implements IMediatorImpl, AfterViewInit
{
   ...существующий код...
   ngAfterViewInit() {
   this.mediator.moveToState(StateType.MainPanelOnly);
}
```

Здесь мы указали, что класс AppComponent реализует интерфейс AfterViewInit. Этот интерфейс определяет одну функцию с именем ngAfterViewInit, которую мы используем для перехода в состояние MainPanelOnly. Функция ngAfterViewInit автоматически вызывается Angular после инициализации начального представления компонента. Это означает, что Angular уже проанализировал HTML-код нашего компонента, создал все дочерние представления и отобразил HTML-код в браузере. Только на этом этапе у нас есть ссылка на наше дочернее представление SideNavComponent и представление RightScreenComponent, необходимое для Mediator.

Наше приложение теперь загружается и находится в правильном начальном состоянии.

Реагирование на события DOM

Мы почти закончили нашу реализацию шаблонов State и Mediator. Последняя часть головоломки – подключение DOM-событий click, чтобы вызвать изменение состояния. Давайте изменим функцию `buttonClickedDetail`:

```
buttonClickedDetail() {
   this.mediator.moveToState(StateType.DetailPanel);
}
```

Функция `buttonClickedDetail` вызывается, когда пользователь нажимает кнопку **Detail** на нашей главной панели. Все, что теперь нужно сделать этому обработчику событий, – это вызвать функцию moveToState, чтобы перейти в состояние `DetailPanel`. В самом деле очень просто.

Нам также нужно изменить функцию обработчика событий, которая вызывается, когда пользователь находится на панели подробностей, и нажать кнопку <, чтобы вернуться на главную панель. Помните, что мы подключили класс `EventEmitter` в файле `RightScreenComponent` к обработчику событий в классе `AppComponent` с именем `onNotifyRightWindow`. Теперь мы можем изменить этот обработчик:

```
onNotifyRightWindow(message:string):void {
   this.mediator.moveToState(
      this.mediator.getCurrentMainPanelState());
}
```

Здесь мы всего лишь переходим к предыдущему состоянию основной панели. Опять же, очень просто.

Последнее взаимодействие с пользователем, которое нам необходимо обработать, – когда пользователь нажимает кнопку **Показать**/**Скрыть** боковую панель навигации. Эта кнопка будет либо показывать, либо скрывать боковую панель навигации. Помните, что эффект от нажатия на эту кнопку будет немного отличаться в зависимости от того, открыта боковая панель навигации в данный момент или закрыта. Поэтому класс `AppComponent` не должен принимать это решение, поскольку он основан на текущем состоянии.

Тогда имеет смысл просто отловить это событие из нашего класса `AppComponent`, а затем направить принятие решения в класс `Mediator`, поскольку в нем содержится вся информация, необходимая для нашего текущего состояния.

Наш обработчик событий в классе `AppComponent` выглядит так:

```
showHideSideClicked() {
   this.mediator.showHideSideNavClicked();
}
```

Здесь мы просто вызываем функцию showHideSideNavClicked, которая реализуется так:

```
showHideSideNavClicked() {
   switch (this._currentState.getStateType()) {
      case StateType.MainPanelWithSideNav:
         this.moveToState(StateType.MainPanelOnly);
         break;
      case StateType.MainPanelOnly:
         this.moveToState(StateType.MainPanelWithSideNav);
         break;
   }
}
```

Эта функция просто запрашивает объект _currentState и переключается в состояние MainPanelOnly или MainPanelWithSideNav соответственно.

Есть еще одна заключительная вещь, которую мы должны сделать, чтобы завершить реализацию шаблонов State и Mediator, – установить начальное состояние приложения. Это можно сделать, вызвав функцию Mediator, moveToState, когда класс AppComponent был создан и визуализировал себя в DOM. Чтобы перехватить это событие, Angular предоставляет интерфейс AfterViewInit, который реализуется функцией ngAfterViewInit. Нам потребуется обновить определение класса AppComponent для реализации интерфейса, а также для обеспечения реализации:

```
export class AppComponent implements IMediatorImpl,
AfterViewInit {
   ...существующий код...
   ngAfterViewInit() {
      this.mediator.moveToState(StateType.MainPanelWithSideNav);
   }
```

Здесь мы добавили интерфейс AfterViewInit в список интерфейсов, которые реализует класс AppComponent. Мы также предоставили реализацию функции ngAfterViewInit, которая просто вызывает функцию moveToState с желаемым начальным состоянием, которое в этом случае является StateType.MainPanelWithSideNav.

Наша реализация шаблонов State и Mediator завершена.

Резюме

В этой главе мы подробно рассмотрели создание приложения Angular с нуля. Мы поэкспериментировали с дизайном перехода страниц слева направо и узнали, как манипулировать CSS-стилями и переходами для создания визуально привлекательного приложения.

К сожалению, наши первоначальные попытки создания этого приложения закончились большим количеством запутанных и трудно исправляемых локальных переменных, поскольку мы пытались держать все элементы страницы под контролем.

Затем мы сделали шаг назад и обсудили, как шаблоны проектирования State и Mediator могут помочь нам управлять переходами страниц. После этого мы перестроили наше приложение в значимые компоненты и подробно рассмотрели, как применять шаблоны State и Mediator для управления состоянием нашего приложения и сложных переходов страниц.

В следующей главе мы рассмотрим концепцию внедрения зависимости и то, как можно использовать новые возможности языка TypeScript для реализации этой мощной и простой парадигмы объектно-ориентированного проектирования.

Глава 12

Внедрение зависимости

В предыдущей главе мы исследовали концепции объектно-ориентированного программирования и проработали процесс создания приложения в соответствии с принципами объектно-ориентированного проектирования.

Шаблоны проектирования **«Банды четырех»** можно разбить на три основные группы. Первая группа называется **Порождающие шаблоны** и охватывает многочисленные способы создания объектов. Вторая группа называется **Структурные шаблоны** и описывает методы проектирования иерархий объектов, которые работают вместе для достижения инкапсуляции и слабой связанности. Третья группа называется **Поведенческие шаблоны** и предназначена для того, чтобы охватить поведение группы объектов вместе для достижения определенных целей. Со временем эти шаблоны вызывали все больший интерес к поиску простейших и лучших решений для широкого круга задач программирования.

По мере того как системы становились все больше и сложнее, появился ряд шаблонов проектирования поверх оригинального набора от «Банды четырех». Эти шаблоны включают в себя **шаблон модели домена**, **шаблон событий домена**, **шаблон расположения службы** и **шаблон внедрения зависимости** среди прочих. В данной главе мы обсудим шаблоны расположения службы и внедрения зависимости, а также создадим простой инжектор зависимостей. Чтобы начать наше обсуждение, давайте сначала определим, что такое служба.

Служба ориентирована на одну и только одну вещь и основана на принципе единственной ответственности. В качестве примера рассмотрим случай загрузки и гидратации объекта из хранилища данных. Этот объект может быть информацией, относящейся к одному клиенту или массиву клиентов, которые были получены на основе шаблона поиска. Мы бы спроектировали службу для загрузки одной записи о клиенте в клиентский объект, а также спроектировали бы службу для поиска записей о клиентах на основе набора критериев поиска. Каждой из них потребуется еще одна служба, которая несет полную ответственность за возврат соединения с базой данных. Служба соединения с базой данных может полагаться на другую службу, которая считывает файл конфигурации системы, чтобы загрузить параметры конфигурации для множества частей системы. У этих служб, таким образом, есть дерево зависимостей, что означает, что система должна загрузить и запустить некоторые службы, прежде чем она сможет загружать и запускать другие. Следовательно, службы позволяют нам писать слабосвязанные системы.

Имея это в виду, есть два шаблона проектирования, которые могут помочь нам найти и использовать службы в большом приложении. Первый из них называется Service Location, где мы создаем центральный **реестр** доступных служб, а затем запрашиваем эти службы по мере необходимости. Вторым из них является расширение шаблона Service Location, называющееся внедрение зависимости. Благодаря внедрению зависимости вместо запроса доступных служб эти службы автоматически внедряются в наш код, прежде чем они нам понадобятся, и поэтому готовы к использованию.

В этой главе мы начнем с примера того, как шаблон Service Location помогает нам находить нужную нам информацию по мере необходимости. Мы создадим локатор служб и покажем, как этот шаблон используется для того, чтобы сделать наш код более модульным и слабо связанным. Затем мы обсудим недостатки шаблона «Локатор служб» и то, почему внедрение зависимости является лучшим шаблоном для использования в наших случаях. Наконец, мы создадим собственный фреймворк внедрения зависимости, используя декораторы TypeScript.

В этой главе подробно обсуждаются концепции внедрения зависимости, его происхождение из шаблона «Локатор служб» и способы создания вышеупомянутого фреймворка с использованием TypeScript. Если вы используете фреймворк, такой как Angular, вы обнаружите, что в нем уже есть сложный интегрированный фреймворк внедрения зависимости, и как таковой вы сможете использовать его как естественное расширение своего кода. Создавая такой фреймворк с нуля, мы расширим наши знания относительно того, как TypeScript и концепции объектно-ориентированного программирования могут быть использованы для решения некоторых довольно сложных задач.

В этой главе мы рассмотрим следующие темы:

- зависимость объектов;
- Service Location;
- разрешение интерфейса;
- внедрение конструктора;
- внедрение декоратора;
- внедрение зависимости.

Отправка почты

Чтобы приступить к обсуждению внедрения зависимости, давайте создадим простое приложение Node, которое отправляет электронное письмо. Отправка почты является общим требованием большинства систем и, как правило, одним из первых вариантов использования, который необходимо создать, чтобы пользователи могли зарегистрироваться на веб-сайте. Еще до того, как пользователь зашел на ваш сайт, необходимо зарегистрировать его регистрационные данные, и часть

этого процесса обычно включает в себя проверку адреса электронной почты. Итак, давайте рассмотрим, как отправить электронное письмо с помощью Node.

Использование nodemailer

Существует множество пакетов на основе Node, которые мы можем импортировать, чтобы иметь возможность работать с сообщениями электронной почты. В этой главе мы будем использовать пакет `nodemailer`, который можно установить так:

```
npm install --save nodemailer
```

После установки нам понадобится несколько файлов объявлений. Используем `@types`:

```
npm install @types/node --saveDev
npm install @types/nodemailer --saveDev
npm install @types/nodemailer-direct-transport --save
npm install @types/nodemailer-smtp-transport --save
npm install @types/nodemailer-bluebird --save
```

После установки пакета `nodemailer` и соответствующих файлов объявлений TypeScript мы можем следовать примерам, приведенным на сайте Nodemailer, и отправить электронное письмо в три простых шага. Давайте создадим файл `NodeMailer.ts`:

```
import * as nodemailer from 'nodemailer';

var transporter = nodemailer.createTransport(
   `smtp://localhost:1025`
);

var mailOptions : nodemailer.SendMailOptions = {
   from : 'from_test@gmail.com',
   to : 'to_test@gmail.com',
   subject : 'Hello',
   text: 'Hello from node.js'
};

transporter.sendMail( mailOptions, (error, info) => {
   if (error) {
      return console.log(`error: ${error}`);
   }
   console.log(`Message Sent ${info.response}`);
});
```

Здесь мы импортировали модуль `nodemailer`, а затем настроили переменную `transporter`, которая использует SMTP-сервер, найденный на порту `1025` ло-

кального хоста. Как только у нас есть соединение с SMTP-сервером, мы устанавливаем переменную `mailOptions`, которая содержит детали нашего сообщения электронной почты, такие как отправитель, получатель, тема и тело письма. Эти свойства называются `from`, `to`, `subject` и `text` соответственно. И наконец, вызов функции `sendMail` для переменной `transporter` отправит фактическое письмо.

Выполнение кода на этом этапе приведет к ошибке, так как SMTP-сервер не будет найден.

Использование локального SMTP-сервера

Существует несколько реализаций локальных SMTP-серверов, которые мы можем использовать в целях разработки. Если вы работаете в среде Windows, обратите внимание на **Papercut**. Papercut – это простой автономный исполняемый файл, который можно запустить для работы в качестве локального SMTP-сервера. Если вы предпочитаете решения на основе Node, можете использовать `smtp-sink`. Это простой пакет, который также предоставляет локальный SMTP-сервер. Чтобы установить `smtp-sink`, достаточно выполнить следующую команду:

```
npm install -g smtp-sink
```

После установки его можно запустить, просто набрав:

```
smtp-sink
```

Параметры по умолчанию для smtp-sink запускают SMTP-сервер на порту `1025`, а веб-сервер – на порту `1080`, где можно просматривать электронную почту, указав в браузере: `http://localhost: 1080/emails`.

После запуска `smtp-sink` наше приложение сможет отправлять электронную почту на локальный SMTP-сервер.

Служебный класс

Код, который мы создали для отправки электронного письма, может быть преобразован в единый класс, который инкапсулирует весь код установки. Вот как выглядит файл `MailService.ts`:

```
import * as nodemailer from 'nodemailer';

export class MailService {
   private _transporter: nodemailer.Transporter;
   constructor() {
      this._transporter = nodemailer.createTransport(
         `smtp://localhost:1025`
      );
```

```
    }
  sendMail(to: string, subject: string, content: string) {
    let options = {
      from: 'from_test@gmail.com',
      to: to,
      subject: subject,
      text: content
    }
    this._transporter.sendMail(
      options, (error, info) => {
        if (error) {
          return console.log(`error: ${error}`);
        }
        console.log(`Message Sent ${info.response}`);
      });
    }
  }
```

Здесь мы создали класс с именем `MailService`, который инкапсулирует внутреннюю работу пакета `nodemailer` и показывает только простую функцию с именем `sendMail`. Эта функция также уменьшила количество параметров, которые нам нужны, чтобы отправить письмо. Обратите внимание, что мы удалили параметр `from` в пользу жесткого кодирования адреса отправителя внутри класса. Это гарантирует, что все письма, отправленные из нашего приложения, будут приходить с одного и того же адреса электронной почты. Мы будем решать проблему жесткого кодирования письма отправителя чуть позже, но, по крайней мере, эта часть информации теперь централизована в одном месте.

Мы можем использовать этот класс следующим образом:

```
import MailService from './app/MailService';

let gmailService = new MailService();

gmailService.sendMail(
  '<test_user>@gmail.com',
  'Hello',
  'Hello from gmailService');
```

Здесь мы просто создали экземпляр класса `MailService` и вызвали функцию `sendMail` для отправки простого электронного письма.

На данный момент наш класс `MailService` работает как положено и отправляет электронные письма правильно. К сожалению, вызов функции `sendMail` в настоящее время не обеспечивает никакой обратной связи с кодом вызова. Было бы намного лучше, если бы функция `sendMail` предоставляла механизм, позволяющий нам знать, была почта отправлена правильно или нет. Поэтому мы должны

реорганизовать эту функцию, чтобы отобразить результаты фактического вызова для отправки электронного письма:

```
sendMail(to: string, subject: string, content: string)
: Promise<void>
{
   let options = {
      from: '<fromaddress>@gmail.com',
      to: to,
      subject: subject,
      text: content
   }

   return new Promise<void> (
      (resolve: (msg: any) => void,
         reject: (err: Error) => void) => {
            this._transporter.sendMail(
               options, (error, info) => {
                  if (error) {
                    console.log(`error: ${error}`);
                    reject(error);
                  } else {
                    console.log(`Message Sent ${info.response}`);
                    resolve(`Message Sent ${info.response}`);
                  }
            })
      }
   );
}
```

Здесь мы изменили сигнатуру функции `sendMail`, чтобы она возвращала объект `Promise`.

Реализация этого объекта, по сути, оборачивает вызов `this._transporter.sendMail` в новый объект `Promise` и вызывает либо обратный вызов `reject`, если есть ошибка, либо обратный вызов `resolve`, если электронное письмо было отправлено правильно.

Возвращая объект `Promise`, мы теперь можем определить результат письма:

```
gmailService.sendMail(
   "test2@test.com",
   "subject",
   "content").then( (msg) => {
      console.log(`sendMail result :(${msg})`);
} );
```

Здесь мы просто использовали свободный синтаксис и вызвали `then` для объекта `Promise` для выполнения функции после завершения функции `sendMail`.

Настройки конфигурации

При написании кода, отправляющего электронные письма, имеет смысл использовать разные настройки для ваших почтовых сервисов в зависимости от среды развертывания. Когда разработчики работают с кодом электронной почты, они должны иметь возможность использовать локальный SMTP-сервер, чтобы быстро проверять электронную почту, отправляемую на и из разных учетных записей, без фактической отправки электронной почты. В среде тестирования тестировщики должны иметь возможность указать, какие учетные записи они хотят использовать в качестве почтового аккаунта и какой SMTP-сервер использовать. В среде **заводского приемочного тестирования** эти настройки электронной почты могут измениться еще раз, поэтому электронные письма из любой тестовой среды не влияют на среду заводского приемочного тестирования. Окончательные настройки будут, разумеется, установлены для производственной среды.

Изменение настроек в зависимости от того, где развернут код, является распространенной проблемой, которая обычно решается с помощью файла конфигурации определенного типа. Значения конфигурации считываются из файла на диске и используются во всей системе. Различные среды используют различные файлы конфигурации, и системный код не нужно менять, просто чтобы изменить эти настройки.

В наших примерах в настоящее время есть два значения, которые являются хорошими кандидатами для настроек конфигурации. Это строка подключения к SMTP-серверу и адрес электронной почты, с которого отправляются все письма.

Эти настройки могут быть легко выражены в виде интерфейса:

```
export interface ISystemSettings {
   SmtpServerConnectionString: string;
   SmtpFromAddress: string;
}
```

Здесь интерфейс `ISystemSettings` определяет два свойства, которые необходимо изменить при изменении среды. Свойство `SmtpServerConnectionString` будет использоваться для подключения к SMTP-серверу, а свойство `SmtpFromAddress` будет использоваться для указания исходного адреса для всех электронных писем.

Теперь мы можем изменить класс `MailService`, чтобы использовать этот интерфейс:

```
import * as nodemailer from 'nodemailer';
import {ISystemSettings} from './ISystemSettings';
```

```
export class MailService {
   private _transporter: nodemailer.Transporter;
   private _settings: ISystemSettings;
   constructor(settings: ISystemSettings) {
      this._settings = settings;
      this._transporter = nodemailer.createTransport(
         this._settings.SmtpServerConnectionString
      );
   }

sendMail(to: string, subject: string, content: string):
   Promise<void> {
   let options: nodemailer.SendMailOptions = {
      from: this._settings.SmtpFromAddress,
      to: to,
      subject: subject,
      text: content
   }
...существующий код...
```

Здесь мы импортировали интерфейс `ISystemSettings`, создали локальную переменную `_settings` для хранения этой информации и изменили нашу функцию-конструктор, чтобы принимать экземпляр объекта, который реализует интерфейс `ISystemSettings`.

Интерфейс `ISystemSettings` используется в двух местах. Во-первых, когда мы вызываем функцию `nodemailer.createTransport`, то используем свойство `SmtpServerConnectionString`. Во-вторых, когда мы создаем объект `options`, то используем свойство `SmtpFromAddress`.

Использование класса `MailService` теперь означает, что мы должны предоставить оба этих параметра при создании объекта:

```
let mailService = new MailService({
   SmtpServerConnectionString : 'smtp://localhost:1025',
   SmtpFromAddress : 'smtp_from@test.com'
});

mailService.sendMail(
   "test2@test.com",
   "subject",
   "content").then( (msg) => {
      console.log(`sendMail result :(${msg})`);
} );
```

Здесь мы создали объект, который соответствует интерфейсу `ISystemSettings`, то есть у него есть свойства `SmtpServerConnectionString` и `SmtpFromAddress`. Этот объект затем передается в конструктор `GMailService`.

Зависимость объектов

Изменения в классе `MailService` ввели зависимость объектов. Чтобы класс `MailService` мог функционировать, он теперь зависит от экземпляра класса, который обеспечивает реализацию интерфейса `ISystemSettings`. Следовательно, `mailService` зависит от класса, который реализует интерфейс `ISystemSettings`.

Эта зависимость – на самом деле хорошая вещь. Она означает, что мы можем предоставить различные версии классов, которые реализуют интерфейс `ISystemSettings`, без внесения каких-либо изменений в код `MailService`. Это позволяет нам конфигурировать среду, в которой работает класс `MailService`, будь то разработка, тестирование, заводское приемочное тестирование или производство.

Это также позволяет нам тестировать некоторые граничные условия. Другими словами, что происходит, если SMTP-сервер не работает или настроен неправильно? Класс `MailService` правильно сообщает, что произошла ошибка? Какие действия должен предпринять наш код, когда служба не может правильно отправить письмо?

Service Location

Наша текущая реализация класса `MailService` использует код вызова для создания экземпляра интерфейса `ISystemSettings` и передачи его в конструктор.

Когда мы пишем код, который создает экземпляр класса `MailService`, мы вынуждены предоставлять интерфейс `ISystemSettings` во время создания. Это зависимость во время компиляции. Другими словами, изменение экземпляра `ISystemSettings` требует изменений в исходном коде, а затем перекомпиляции. Однако было бы намного лучше, если бы мы установили эти параметры во время выполнения.

Для этого класс `MailService` должен запрашивать экземпляр класса, реализующего интерфейс `ISystemSettings`, во время выполнения, а не во время компиляции.

Если сам класс запрашивает конкретный объект, который в настоящее время реализует интерфейс, то этот процесс называется **Service Location**. Другими словами, сам класс пытается найти службу, обеспечивающую реализацию интерфейса.

Однако для того, чтобы это работало, нам нужен центральный реестр, который может ответить на следующий вопрос: дайте мне конкретный класс, который в настоящее время реализует этот интерфейс. В этом заключается суть шаблона проектирования Service Location.

Давайте создадим простой класс, который реализует шаблон проектирования Service Location. Для этого нам понадобится класс локатора служб, который дол-

жен будет сделать две вещи. Во-первых, он должен предоставить механизм для регистрации реализаций класса в интерфейсе. Во-вторых, он должен предоставить классу механизм для выполнения текущей реализации интерфейса.

Мы можем реализовать простой локатор служб следующим образом:

```
import {ISystemSettings} from './SystemSettings';

export type IRegisteredClasses = ISystemSettings | undefined;

export class ServiceLocator {
   static registeredClasses: Map<string, IRegisteredClasses>;
   static initialised: boolean = false;

   static init() {
      this.registeredClasses =
      new Map<string, IRegisteredClasses>();
      this.initialised = true;
}

public static register(interfaceName: string, instance:
   IRegisteredClasses) : void {
   if (!this.initialised) {
      this.init();
   }
   this.registeredClasses.set(interfaceName, instance);
}

public static resolve(interfaceName: string):
   IRegisteredClasses {
      return this.registeredClasses.get(interfaceName);
   }
}
```

Здесь мы начинаем с определения псевдонима типа `IRegisteredClasses`, который позволит нам описать, какие интерфейсы могут быть зарегистрированы в локаторе служб. Затем мы определяем класс, который является самим локатором служб, `ServiceLocator`. У него есть два внутренних свойства, `registeredClasses` и `initialised`. Свойство `registerClasses` используется для хранения экземпляров каждого зарегистрированного класса в `Map`. Свойство `initialised` используется просто, чтобы убедиться, что свойство `Map` создано правильно перед первым использованием, в рамках функции `init`.

Функция `register` принимает два параметра – `interfaceName` типа `string` и экземпляр класса типа `IRegisteredClasses`. Функция `register` добавляет экземпляр класса к `this.registeredClasses`, используя параметр `interfaceName` в качестве ключа. Функция `resolve` возвращает экземпляр класса на основе параметра `interfaceName`, который передается как ключ.

Этот очень простой класс `ServiceLocator` можно затем использовать так:

```
import {ServiceLocator, IRegisteredClasses} from
   './ServiceLocator';
import {ISystemSettings} from './SystemSettings';

let settings: ISystemSettings = {
   SmtpServerConnectionString: ``,
   SmtpFromAddress: `from_test@test.com`
}

ServiceLocator.register('ISystemSettings', settings);

let currentSettings: IRegisteredClasses =
   ServiceLocator.resolve(`ISystemSettings`);

if (currentSettings) {
   console.log(`SmtpFromAddress :
      ${currentSettings.SmtpFromAddress}`);
} else {
   console.log(`currentSetting is undefined.`);
}
```

Здесь мы создали экземпляр объекта для предоставления двух свойств, необходимых интерфейсу `ISystemSettings`, и назвали его `settings`. Затем мы вызываем функцию `register`, чтобы зарегистрировать этот объект с помощью ключа `'ISystemSettings'`. Как только объект зарегистрирован, мы можем вызвать функцию `resolve` класса `ServiceLocator` для извлечения текущего зарегистрированного объекта для этого ключа. После этого мы выводим результаты в консоль.

Теперь мы можем обновить класс `MailService`, чтобы использовать класс `ServiceLocator`:

```
export class MailService {
   private _transporter: nodemailer.Transporter | undefined;
   private _settings: ISystemSettings | undefined;
   constructor() {
      this._settings = ServiceLocator.resolve(`ISystemSettings`);
      if (this._settings) {
         this._transporter = nodemailer.createTransport(
            this._settings.SmtpServerConnectionString);
      }
   }
   ...существующий код...
```

Здесь мы обновили функцию-конструктор класса `MailService`, чтобы использовать шаблон локатора служб. Наше внутреннее свойство `_settings` по-прежнему

содержит экземпляр объекта `ISystemSettings`, но сам класс `MailService` запрашивает экземпляр интерфейса `ISystemSettings` из класса `ServiceLocator`.

Теперь мы можем создать экземпляр класса `MailService`:

```
let gmailService = new MailService();
```

Обратите внимание, что мы скрыли внутренние зависимости класса `MailService` от пользователя класса, используя шаблон локатора служб. Сам класс запрашивает ресурсы, которые ему необходимы для выполнения своих функций.

Антишаблон Service Location

Идеи, лежащие в основе шаблона «Локатор служб», были впервые представлены Мартином Фаулером в 2004 году в блоге под названием *«Инверсия контейнеров управления и шаблон внедрения зависимостей»* (`http://martinfowler.com/articles/injection.html`). С тех пор этот шаблон был создан и испытан в полевых условиях в разных языках и средах. В своей книге *«Внедрение зависимости в .NET»* Марк Симан утверждает, что шаблон Service Location фактически является антишаблоном.

По мнению Марка, слишком легко неправильно понять использование определенного класса, когда применяется Service Location. В крайнем случае, каждая функция класса может использовать разные службы, что означает, что пользователь класса должен прочитать всю кодовую базу, чтобы понять, какие зависимости есть у класса.

Внедрение зависимости

Марк Симан утверждает, что лучший способ использования Service Location – перечислить все зависимости класса в конструкторе класса, а затем передать процесс создания класса чему-то, что понимает, как разрешить все эти зависимости. Процесс создания класса можно рассматривать как сборку экземпляра класса и заполнение доступных служб.

Таким образом, когда запрашивается экземпляр класса, разрешаются зависимости класса, и процесс ассемблера просто дает нам экземпляр класса, который работает правильно. Другими словами, все зависимости, которые есть у класса, внедряются в класс ассемблером до того, как класс будет предоставлен нам.

В этом суть шаблона проектирования внедрения зависимости.

Создание инжектора зависимостей

В этом разделе мы будем использовать знания, полученные нами при написании локатора служб, и объединять их с декораторами TypeScript для создания простого фреймворка, реализующего шаблон внедрения зависимости. Однако, прежде чем сделать это, давайте обсудим проблему разрешения интерфейса.

Разрешение интерфейса

Как мы знаем, ключевое слово `interface` – это конструкция языка TypeScript, которую мы используем для определения формы классов или объектов. Везде, где нам нужно определить пользовательский тип и нужен компилятор TypeScript, чтобы гарантировать, что свойства и функции доступны для объекта, мы используем интерфейс. Интерфейсы особенно удобны при описании служб, где любое количество служб может предоставлять такую же функциональность нашему коду. Чтобы создать полезный инжектор зависимостей, нам нужно суметь ответить на следующий вопрос: учитывая интерфейс, как можно получить службу, которая в настоящее время реализует его?

В нашей текущей реализации Service Location мы просто используем строковые значения для регистрации (`register`) и выполнения (`resolve`) интерфейса, как показано в двух вызовах класса `ServiceLocator`. `register` выглядит следующим образом:

```
ServiceLocator.register('ISystemSettings', smtpSinkSettings);
```

А `resolve` – так:

```
this._settings = ServiceLocator.resolve('ISystemSettings');
```

К сожалению, в этих случаях следует избегать использования строк. Слишком легко неправильно набрать саму строку и в результате ввести ошибки времени выполнения. Опять же, мы не можем использовать само имя интерфейса в этом случае, так как интерфейсы компилируются в результирующем JavaScript-коде.

Разрешение enum

Как мы видели в предыдущих главах, магические строки являются ярким примером, где можно реорганизовать наш код для использования разрешения enum. В качестве примера рассмотрим класс `ServiceLocator`, построенный на основе этого разрешения:

```
interface ISystemSettings {
}
```

```
interface IMailService {
}

enum Interfaces {
   ISystemSettings,
   IMailService
}

class ServiceLocatorTypes {
   public static register(
      interfaceName: Interfaces, instance: any) {}
      public static resolve( interfaceName: Interfaces) {}
}

ServiceLocatorTypes.register(Interfaces.ISystemSettings, {});

ServiceLocatorTypes.resolve(Interfaces.ISystemSettings);
```

Здесь вначале идут два интерфейса, которые мы хотим использовать с нашим локатором служб, `ISystemSettings` и `IMailService`. Обратите внимание, что мы исключили внутренние свойства этих интерфейсов, чтобы упростить обсуждаемый код.

Далее мы определили перечисление с именем `Interfaces`, которое содержит запись для каждого из интерфейсов, которые мы хотим использовать. Наше определение класса (опять-таки без реализаций функций) для класса `ServiceLocatorTypes` просто показывает изменение в сигнатурах функций `register` и `resolve` для использования перечисления `Interfaces`.

В последних двух строках этого фрагмента кода показано, как будет использоваться перечисление `Interfaces` при вызове функций `register` и `resolve`. Используя перечисление для хранения имен интерфейсов, мы исключили использование магических строк, и теперь у нас есть основное перечисление для описания всех интерфейсов, которые будут использоваться системой.

Разрешение класса

В качестве альтернативы реализации `enum` мы также можем использовать классы специального назначения. Это лучше всего проиллюстрировать, посмотрев на приведенный ниже пример кода:

```
interface ISystemSettings { }
class IISystemSettings { }

interface IMailService { }
class IIMailService { }

class ServiceLocatorGeneric {
```

```
    public static register<T>(
      interfaceName: {new(): T;}, instance: any) {}
    public static resolve<T>(
      interfaceName: {new() : T}) {}
}

ServiceLocatorGeneric.register(IISystemSettings, {});

ServiceLocatorGeneric.resolve(IISystemSettings);
```

Вначале идет интерфейс `ISystemSettings`, который мы хотим использовать с нашим локатором служб. Затем мы определяем класс `IISystemSettings`, у которого нет функций или свойств, и он используется только для разрешения интерфейса. Наименование этого класса важно. По соглашению мы назвали этот класс так же, как и интерфейс, который мы описываем, но добавили дополнительную букву «I» в начало имени. Это означает, что интерфейс с именем `ITest` будет иметь соответствующий класс `ITest`, единственной целью которого является предоставление уникального имени (вместо перечисления) при использовании с фреймворком, реализующим шаблон внедрения зависимости.

Наш класс `ServiceLocatorGeneric` также изменил сигнатуры функций `register` и `resolve`, чтобы приспособить использование имени класса вместо перечисления. Теперь мы используем синтаксис обобщений и требуем, чтобы аргумент `interfaceName` имел тип `{new(): T;}`. Вы помните из нашего обсуждения обобщений, что при использовании функции, которая должна вызывать функцию `new()` для создания экземпляра класса, когда есть имя класса, к ней должен обращаться конструктор класса.

Давайте посмотрим на реализацию функции `register`:

```
public static register<T>(
   t: {new(): T},
   instance: IRegisteredClassesGeneric) : void
{
   if (!this.initialised) { this.init();
   }
   let interfaceInstance = new t();
   let interfaceName = interfaceInstance.constructor.name;
   console.log(`ServiceLocator registering : ${interfaceName}`);

   this.registeredClasses.set(interfaceName, instance);
}
```

Здесь мы обновили нашу функцию `register`, чтобы использовать синтаксис обобщений с помощью типа `T`.

Первый параметр функции называется `t` и использует синтаксис `t:{new():T}`, чтобы позволить нам создать объект типа `T` в нашем коде. После проверки, была

ли вызвана функция `init`, мы создаем экземпляр этого обобщенного класса. После этого мы можем узнать имя этого класса, запросив свойство `constructor.name`. Это свойство будет использоваться вместо реализации волшебных строк или перечисления, которые мы использовали ранее.

Синтаксис обобщений затем позволяет нам вызывать функции `register` и `resolve`, просто предоставляя имя класса, как видно из последних двух строк предыдущего фрагмента кода. Если мы сравним разрешение в стиле `enum` со стилем разрешения имен классов, то получим следующие фрагменты кода. Разрешение в стиле `enum` выглядит так:

```
ServiceLocatorTypes.register(Interfaces.ISystemSettings, {});
```

Разрешение имен классов демонстрируется следующим образом:

```
ServiceLocatorGeneric.register(IISystemSettings, {});
```

В оставшейся части этой главы мы будем использовать разрешение имен классов по ряду причин.

Определение интерфейса и имя класса, используемого для разрешения интерфейса, определены в одном и том же исходном файле. При использовании разрешения в стиле `enum` определения интерфейса разбросаны по всей базе кода, а экземпляр `enum` находится в одном файле. Это дает нам возможность изменить код в двух местах, когда необходим новый интерфейс, который будет использоваться в Service Location.

Использование определений классов означает, что нужно вводить меньше кода. Хотя это может показаться тривиальной причиной, это также означает, что нужно читать меньше кода. Будучи разработчиками, мы проводим весь день за чтением и написанием кода, и чем меньше нам нужно читать, чтобы донести сообщение, тем лучше.

Стандарт двойного именования интерфейса `II` представляет собой визуальный триггер, который указывает, что этот код использует Service Location. Всякий раз, когда мы читаем код и видим двойной префикс `II`, мы сразу же узнаем, что Service Location в игре. Это помогает нам довольно быстро отличать стандартные интерфейсы от интерфейсов на основе Service Location.

Теперь давайте кратко рассмотрим функцию `resolve` класса `ServiceLocatorGeneric`:

```
public static resolve<T>(t: { new(): T }):
   IRegisteredClassesGeneric {
   let interfaceInstance = new t();
   let interfaceName = interfaceInstance.constructor.name;
   console.log(`ServiceLocator resolving : ${interfaceName}`);
   return this.registeredClasses.get(interfaceName); }
```

Здесь мы добавили обобщенный тип T к функции `resolve` и снова создали новый экземпляр класса типа `T`, чтобы узнать свойство `constructor.name`. Затем мы используем его, чтобы найти класс, зарегистрированный для интерфейса.

Внедрение конструктора

Ранее мы обсуждали преимущества и антишаблоны, применяемые при использовании шаблона локатора службы, и подхватили идеи Марка Симана, согласно которым внедрение зависимости должно встречаться только в конструкторах классов. Мы использовали Service Location в классе `MailService` в функции-конструкторе:

```
export default class MailService {
   private _transporter: nodemailer.Transporter;
   private _settings: ISystemSettings;

   constructor() {
      this._settings = ServiceLocator.resolve('ISystemSettings');
      this._transporter = nodemailer.createTransport(
         this._settings.SmtpServerConnectionString
      );
   }
```

Здесь мы указали локальное свойство `_settings` типа `ISystemSettings` и используем `ServiceLocator` для выполнения (`resolve`) этого внутреннего свойства. Переключение на шаблон внедрения зависимости с использованием инжектора конструктора будет выглядеть так:

```
export default class MailServiceDi {
   private _transporter: nodemailer.Transporter;
   private _settings: ISystemSettings;

   constructor(_settings?: IISystemSettings) {
      this._transporter = nodemailer.createTransport(
         this._settings.SmtpServerConnectionString
      );
   }
```

Есть несколько моментов, на которые следует обратить внимание. Во-первых, у нас все еще есть закрытое свойство `_settings`, которое вводится в интерфейс `ISystemSettings`. Это означает, что мы все еще можем ссылаться на `this._settings` в теле кода. Во-вторых, теперь мы включили в наш конструктор параметр `_settings ?: IISystemSettings`. Поэтому мы ожидаем, что инжектор зависимостей найдет реализацию интерфейса `ISystemSettings`, или, точнее, класса, который зарегистрирован для ключа `IISystemSettings`, и внедрит это в наш класс, чтобы закрытое свойство `_settings` содержало эту реализацию.

Чтобы наш инжектор зависимостей работал, имя параметра конструктора и имя закрытого свойства должны быть одинаковыми.

Давайте посмотрим, как будет выглядеть результат внедрения конструктора после того, как сам класс обработан фреймворком инжектора зависимостей:

```
export default class MailServiceDi {
   private _transporter: nodemailer.Transporter;
   // private _settings: ISystemSettings;
   get _settings() : ISystemSettings {
      return ServiceLocatorGeneric.resolve(IISystemSettings);
   }

   constructor(_settings?: IISystemSettings) {
      this._transporter = nodemailer.createTransport(
         this._settings.SmtpServerConnectionString
      );
   } }
```

Здесь у нас пример того, как должен выглядеть класс после внедрения. Закрытое свойство `_settings` было заменено функцией `get` с тем же именем, то есть `get _settings()`. Эта функция внутренне вызывает наш класс `ServiceLocator` для выполнения (resolve) интерфейса. Создав простую функцию get, мы, по сути, внедрили нашу зависимость. Однако это поднимает другой вопрос. Как можно изменить класс во время выполнения, чтобы внедрить необходимые ему зависимости и создать функции `get`? Ответ – с помощью декораторов.

Внедрение декораторов

В начале `главы 4` *«Декораторы, обобщения и асинхронные функции»* мы обсудили использование декораторов и то, как они вызываются при определении класса. Декораторы не вызываются, когда создается экземпляр класса, поэтому их использование ограничено опросом и манипулированием определениями класса. Декораторы, как мы знаем, могут применяться к классам, свойствам, функциям и параметрам. Давайте создадим простой декоратор класса и посмотрим, какую информацию о классе он нам дает.

Помните, что в классе есть три вещи, которые нас интересуют во время выполнения этого упражнения. Во-первых, нам нужно найти определение конструктора класса. Как только мы узнаем, как выглядит конструктор, нам нужно найти список параметров, которые использует конструктор. Каждый из этих параметров затем станет методом чтения, который использует наш локатор служб для разрешения зависимостей. Последний фрагмент информации, который нам понадобится, – это найти тип, ожидаемый каждым параметром конструктора. Получив эту информацию, мы можем создать простую функцию-геттер, которая будет возвращать правильный тип в нашем декораторе.

Использование определения класса

Прежде чем мы начнем, имейте в виду, что для использования декораторов наш файл `tsconfig.json` должен быть правильно настроен:

```
{
   "compilerOptions": {
      "target": "es6",
      "module": "commonjs",
      "lib": [
        "es6"
      ],
      "strict": true,
      "esModuleInterop": true,
      "experimentalDecorators": true,
      "emitDecoratorMetadata": true,
   }
}
```

Здесь мы включили опцию `"experimentalDecorators": true`, а также опцию `"emitDecoratorMetadata": true`, чтобы наш код использовал декораторы, как мы это делали в `главе 4` *«Декораторы, обобщения и асинхронные функции»*. Мы также определили, что хотим использовать ES6 в качестве цели компиляции, и включили библиотеку es6 в определение `lib`. Нам также необходимо установить пакет reflect-metadata:

```
npm install reflect-metadata
```

Давайте соберем простой декоратор класса и посмотрим, какую информацию мы можем извлечь из класса. Наш декоратор в файле `Decorator.ts` выглядит так:

```
import 'reflect-metadata';
export function ConstructorInject(classDefinition: Function) {
   console.log(`classDefinition:`);
   console.log(`================`);
   console.log(`${classDefinition}`);
   console.log(`================`);
}
```

Здесь мы импортируем модуль `reflect-metadata`, а затем создаем декоратор, который просто записывает значение аргумента `classDefinition` в консоль.

Теперь мы можем декорировать класс `MailServiceDi` с помощью этого декоратора и посмотреть, что произойдет:

```
import { ConstructorInject } from './ConstructorInject';

@ConstructorInject
```

```
export default class MailServiceDi {
   private _transporter: nodemailer.Transporter;
   private _settings: ISystemSettings;
   constructor(_settings?: IISystemSettings) {
   }
}
```

Здесь мы импортировали наш декоратор `ConstructorInject` и применили его к классу `MailServiceDi`. Обратите внимание, что для краткости мы удалили тело кода конструктора, который настраивает свойство `_transporter`. Если мы сейчас создадим экземпляр этого класса, он будет выглядеть так:

```
import MailServiceDi from './MailServiceDi';
var gmailDi = new MailServiceDi();
```

Мы сгенерируем следующий вывод консоли из нашего декоратора `ConstructorInject`:

```
classDefinition:
================
class MailServiceDi {
   constructor(_settings) { }
}
================
```

Как видно, параметр `classDefinition` заполняется полным определением класса для класса `MailServiceDi`. Это определение, однако, не является TypeScript-определением нашего класса. Это определение JavaScript. Это означает, что мы потеряли информацию о типе каждого из наших параметров конструктора, так как эта информация компилируется. Однако у нас есть имя свойств, которые этот класс использует в своем конструкторе.

Сгенерированный код JavaScript всегда будет включать в себя конструктор в качестве первой функции. Если нужно добавить любую другую функцию вверху определения класса и написать конструктор внизу определения класса, TypeScript всегда будет перемещать функцию-конструктор в начало определения класса.

Синтаксический анализ параметров конструктора

Имея доступ к полному определению класса, мы можем использовать простой поиск строки, чтобы найти свойства конструктора класса. Если мы находим символ первой открывающей скобки '`(`' и следующий символ закрывающей скобки '`)`', то можем извлечь строку, содержащую все имена параметров нашего конструктора. Давайте обновим наш код `ConstructorDecorator`:

```
let firstIdx = classDefinition.toString().indexOf('(') + 1;
let lastIdx = classDefinition.toString().indexOf(')');
let arr = classDefinition.toString().substr(
  firstIdx, lastIdx - firstIdx);

console.log(`class parameters :`);
console.log(`${arr}`);
console.log(`==================`);
```

Вывод этого кода выглядит так:

```
================
class parameters :
_settings
==================
```

Мы можем проверить этот код, вставив еще один параметр в наш конструктор и проверив вывод. Итак, если у класса `MailServiceDi` есть два аргумента, соответствующий фрагмент кода будет выглядеть так:

```
constructor(_settings?: IISystemSettings, testParameter?: string) {
}
```

Затем при синтаксическом анализе конструктора будет получен следующий фрагмент кода:

```
class parameters :
_settings, testParameter
==================
```

Таким образом, с помощью простой экстраполяции строк мы можем выяснить, какие имена свойств требуются этому классу. Затем мы можем легко проанализировать этот массив в функции `ConstructorInject`:

```
let splitArr = arr.split(', ');

for (let paramName of splitArr) {
  console.log(`found parameter named : ${paramName}`);
}
```

Здесь мы создаем массив с именем `splitArray` из строки, содержащей имена наших параметров, и записываем каждую запись в консоль. Вывод будет следующим:

```
found parameter named : _settings
found parameter named : testParameter
```

Итак, теперь у нас есть массив, который определяет имена параметров для нашей функции-конструктора.

Поиск типов параметров

Теперь, когда мы знаем имена параметров нашего конструктора, нам нужно сопоставить их с типом параметра. Для этого нам потребуется использовать пакет `reflect-metadata` в декораторе `ConstructorInject`:

```
let parameterTypeArray =
  Reflect.getMetadata("design:paramtypes", classDefinition);

console.log(`parameterTypeArray:`);
console.log(`==================`);
console.log(`${parameterTypeArray}`);
console.log(`==================`);

for (let type of parameterTypeArray) {
  console.log(`found type : ${type.name}`);
}
```

Здесь мы вызываем функцию `Reflect.getMetadata` и используем аргумент `"design: paramtypes"` для извлечения массива из определения класса. Затем мы выводим этот массив в консоль, а после этого перебираем массив, чтобы вывести свойство `name` каждого элемента в массиве `"design: paramtypes"`. Вывод этого кода выглядит так:

```
parameterTypeArray:
==================
class IISystemSettings {
},function String() { [native code] }
==================
found type : IISystemSettings
found type : String
```

Этот тип информации – именно то, что нам нужно для создания инжектора конструктора. Обратите внимание, что первый параметр, который мы знаем, имеет имя `_settings` и тип `IISystemSettings`. Второй параметр `testParameter` имеет тип `String`.

Внедрение свойств

Теперь мы можем объединить результаты обоих массивов, чтобы сопоставить имя параметра с именем типа в коде декоратора:

```
for (let i = 0; i < splitArr.length; i++) {
  let propertyName = splitArr[i];
  let typeName = parameterTypeArray[i];
  console.log(`
    parameterName : ${propertyName}
```

```
        is of type : ${typeName.name}`);
    }
```

Здесь мы перебираем массив `splitArr` (который содержит имена наших параметров) и используем тот же индекс для свойства `parameterTypeArray` для сопоставления имен свойств с именами типов. Результат выглядит следующим образом:

```
parameterName  : _settings
is of type     : IISystemSettings

parameterName  : testParameter
is of type     : String
```

Имея под рукой эту информацию, мы можем теперь использовать JavaScript для внедрения требуемого нам свойства:

```
Object.defineProperty(classDefinition.prototype, propertyName, {
   get: function() {
      return ServiceLocatorGeneric.resolve(
         eval(typeName)
      );
   });
```

Здесь мы используем функцию `Object.defineProperty`, которую JavaScript предоставляет для создания свойства во время выполнения и присоединения его к определению нашего класса. Функция `defineProperty` принимает три параметра. Первый параметр – это прототип класса, который нужно изменить. Второй параметр – это свойство `propertyName`, а третий параметр – это определение свойства. Наше определение свойства – это простая функция-геттер, которая затем вызывает функцию `ServiceLocatorGeneric.resolve`, передавая функцию `typeName`. Обратите внимание, что мы вызвали функцию `eval`, передав ей функцию `typeName`, которую мы извлекли из нашего параметра `TypeArray`. Этот шаг необходим для отправки определения класса в локатор служб вместо простой строки.

Использование внедрения зависимости

Теперь, когда мы внедряем функции свойств через декоратор ConstructorInjector, мы можем использовать наш фреймворк инжектора зависимостей:

```
import GMailServiceDi from './app/GMailServiceDi';
import { ServiceLocatorGeneric } from './app/ServiceLocator';
import { IISystemSettings } from './app/ISystemSettings';

ServiceLocatorGeneric.register(IISystemSettings, {
   SmtpServerConnectionString : 'smtp://localhost:1025',
```

```
    SmtpFromAddress : 'smtp_from@test.com'
});

var gmailDi = new GMailServiceDi();
    gmailDi.sendMail("test@test.com", "testsubject", "testContent"
).then( (msg) => {
    console.log(`sendMail returned : ${msg}`);
} ).catch( (err) => {
    console.log(`sendMail returned : ${err}`);
});
```

После импорта различных модулей в наш образец мы вызываем `ServiceLocatorGeneric.register` для регистрации объекта, предоставляющего интерфейс `IISystemSettings`. Затем мы просто создаем экземпляр класса `MailServiceDi` без параметров. На этом этапе наш инжектор зависимостей выполнил всю работу за нас и внедрил правильные свойства для незамедлительного использования.

Обратите внимание, насколько прост этот конструктор объекта, `new MailServiceDi()`. Он выглядит так же, как и любое другое обычное создание объекта. Как только класс был создан, мы можем вызвать функцию `sendMail`, как делали раньше.

Рекурсивное внедрение

В качестве финального теста нашего фреймворка давайте теперь добавим класс `MailServiceDi` в другой класс. Это означает, что наш новый класс будет зависеть от интерфейса `IMailServiceDi`, который сам по себе зависит от интерфейса `ISystemSettings`. Это пример дерева рекурсивных зависимостей.

Мы начнем с определения интерфейса для самого класса `MailServiceDi`:

```
export interface IMailServiceDi {
    sendMail(to: string, subject: string, content: string)
    : Promise<void>;
}

export class IIMailServiceDi { }
```

Здесь мы взяли определение функции `sendMail`, которая возвращает `Promise`, и создали интерфейс с именем `IMailServiceDi`. Мы также создали класс `IIMailServiceDi`, который будет использоваться в качестве поиска типов нашим фреймворком.

Имея эти интерфейсы, мы можем создать класс, который зависит от интерфейса `IMailServiceDi`:

```
import { IMailServiceDi, IIMailServiceDi } from "./MailServiceDi";
import { ConstructorInject } from "./Decorators";

@ConstructorInject
export class MailSender {
   private mailService: IMailServiceDi | undefined;

   constructor(mailService?: IIMailServiceDi) { }

   async sendWelcomeMail(to: string) {
      if (this.mailService) {
         let response = await this.mailService
         .sendMail(to, "Welcome", "Welcome from MailSender");
         console.log(`MailSender.sendMail returned : ${response}`);
      }
   }
}
```

Здесь мы создали класс с именем `MailSender` и использовали декоратор `ConstructorInject` для декорирования класса. У нашего класса есть закрытое свойство mailService типа `IMailServiceDi`. Это свойство, которое будет создано нашим фреймворком. Функция-конструктор просто использует класс `IIMailService` для указания того, что свойство `mailService` должно быть введено. У класса `MailSender` есть функция `sendWelcomeMail`, которая использует шаблон `async await` для вызова функции `SendMail MailServiceDi`.

Чтобы протестировать этот класс, нам нужно зарегистрировать класс `Mail ServiceDi` в нашем фреймворке, а затем создать новый экземпляр класса `MailSender` и вызвать функцию `sendWelcomeMail`:

```
let mailServiceDi = new MailServiceDi();
ServiceLocatorGeneric.register(IIMailServiceDi, mailServiceDi);
let mailSender = new MailSender();
mailSender.sendWelcomeMail("test@test.com");
```

Вывод этого кода выглядит так:

```
ServiceLocator resolving : IIMailServiceDi
ServiceLocator resolving : IISystemSettings
Message Sent 250
MailSender.sendMail returned : Message Sent : 250
```

Здесь видно, что фреймворк вызывает класс `ServiceLocator` для выполнения (`resolve`) `IIMailServiceDi`. Это во время конструктора класса `MailSender`. Поскольку класс `MailServiceDi` зависит от интерфейса `ISystemSettings`, выполняется второй вызов для выполнения (`resolve`) `IISystemSettings`.

Резюме

В этой главе мы обсудили локаторы служб и шаблоны проектирования внедрения зависимости. Мы начали с создания класса для отправки электронных писем, а затем создали наш собственный простой локатор служб, который мог бы выполнять экземпляры классов, учитывая строковое имя. Затем мы перешли к более решительному шаблону Service Location, который использовал имена классов вместо магических строк в качестве ключа для регистрации и выполнения экземпляров классов. Затем мы обсудили подводные камни шаблона Service Location и реализовали фреймворк внедрения зависимости, используя декораторы.

В следующей главе мы рассмотрим, что нужно для создания приложений, которые объединяют веб-сервер, такой как Node и Express, и дружественный к TypeScript фреймворк, такой как Angular.

Глава 13

Создание приложений

До сих пор мы изучали различные инструменты и методы, которые нам необходимо использовать для создания приложения с применением самых популярных фреймворков TypeScript. Мы сосредоточились на создании сайта с довольно сложным экраном с помощью Angular и узнали, как можно использовать шаблоны проектирования State и Mediator для модульного решения этой сложности. Мы также рассмотрели использование Node и Express, которые позволяют создавать серверное приложение для обработки генерации страниц, и обработку маршрутов. В первом разделе этой главы мы объединяем их и смотрим, как обслуживать сайт на Angular с помощью сервера Node Express. Мы будем опираться на приложение Angular, которое мы создали в `главе 11` *«Объектно-ориентированное программирование»*, и использовать его в качестве основы для дальнейшей разработки.

После того как у нас есть Express, обслуживающий наше приложение на Angular, мы сможем изучить ряд других требований, которые наш веб-сервер должен будет выполнить. Как войти в приложение? Как управлять правами безопасного доступа на сервере и не хранить куки в браузере? Как настроить наш сервер Express таким образом, чтобы он мог быть развернут в различных средах тестирования? Во втором разделе этой главы мы рассмотрим инструменты и методы для создания полноценного сервера Express, который будет отвечать этим требованиям. Затем мы обновим наше приложение на Angular для интеграции с этим сервером. Некоторые из тем, которые мы рассмотрим, включают в себя гарантию того, что наше приложение может обслуживать неаутентифицированных пользователей только на странице входа, а после входа предоставлять полный набор функций.

В последнем разделе этой главы мы рассмотрим интеграцию внешних поставщиков аутентификации, таких как Google, для обеспечения технологии единого входа. Мы рассмотрим следующие темы:

- интеграция Node и Angular;
- настройка сервера Express и ведение журнала сервера;
- использование Brackets для разработки экранов пользовательского интерфейса;
- маршрутизация в Angular и Auth Guard;
- отправка данных в конечную точку Express в Angular;
- использование Observables;

- применение JWT-токенов;
- интеграция с аутентификацией через Google.

Интеграция Node и Angular

Настройка по умолчанию для приложения Angular, настроенного с помощью интерфейса командной строки Angular, уже содержит ряд опций, которые можно использовать в рабочих настройках. Одной из этих опций является свойство `outputPath`, которое указывает, куда записываются скомпилированные файлы JavaScript. Используя отдельный путь для выходных файлов, мы знаем, что можем распространять содержимое этого каталога как полностью рабочую версию нашего сайта.

Опция `outputPath` находится в файле `angular.json` в разделе **Projects | Angular-sample | Architect | Build | Options (Проекты | Образец Angular | Архитектор | Сборка | Опции**). Это свойство по умолчанию установлено как "`dist/angular-sample`", и поэтому, когда мы запустим команду `ng build`, то заметим, что интерфейс командной строки Angular сгенерирует несколько файлов в этом каталоге. К ним относятся `main.js`, `polyfill.js`, `runtime.js`, `styles.js` и `vendor.js`. Эти файлы являются выходными данными шага компиляции Angular. Наряду с этими файлами в этом каталоге также есть файл `index.html`, содержимое которого выглядит так:

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>AngularSample</title>
    <base href="/">
    <meta name="viewport" content="width=device-width,
      initial-scale=1">
    <link rel="icon" type="image/x-icon" href="favicon.ico">
  </head>
  <body>
    <app-root></app-root>
    <script type="text/javascript" src="runtime.js"></script>
    <script type="text/javascript" src="polyfills.js"></script>
    <script type="text/javascript" src="styles.js"></script>
    <script type="text/javascript" src="vendor.js"></script>
    <script type="text/javascript" src="main.js"></script>
  </body>
</html>
```

Этот HTML-файл автоматически генерируется на шаге компилятора Angular и, по сути, содержит теги `<app-root>` и `<script>` для каждого из сгенерированных

файлов JavaScript. Тег `<app-root>` используется в качестве базового элемента DOM, где будут отображаться элементы нашего приложения.

Давайте теперь сгенерируем производственную сборку для нашего приложения, используя интерфейс командной строки Angular:

```
ng build --prod
```

Здесь мы указали опцию `--prod` при использовании `ng build`, которая выполнит несколько дополнительных шагов компиляции для генерации готовой к использованию версии нашего приложения Angular. Затем если мы проверим сгенерированный файл `index.html` в каталоге `dist`, то заметим несколько незначительных изменений:

```
<!doctype html>
<html lang="en">
  <head>
    ...существующий html-код...
    <link rel="stylesheet" href="styles.06e5b622c6ba37e242d1.css">
  </head>
  <body>
    <app-root></app-root>
    <script type="text/javascript"
      src="runtime.06daa30a2963fa413676.js"></script>
    <script type="text/javascript"
      src="polyfills.a5acfd4a5754e593d36f.js"></script>
    <script type="text/javascript"
      src="main.85661e3d3f241cd36aa4.js"></script>
  </body>
</html>
```

Здесь видно, что был создан новый элемент `<link>`, который обращается к CSS-файлу, и что список тегов `<script>` был уменьшен с пяти файлов до трех. Также обратите внимание, что компилятор Angular добавил длинную случайную строку в имена файлов для CSS-файла и файлов JavaScript. Эта строка является частью механизма управления версиями, который Angular использует для того, чтобы при изменении приложения соответствующее имя JavaScript-файла автоматически изменялось, поэтому любой браузер будет вынужден запросить файл повторно.

Интересно отметить, что Angular может изменять содержимое файла `index.html`, который генерируется на основе ряда факторов. Поэтому важно, чтобы мы обслуживали этот файл из каталога `dist`, дабы обеспечить совпадение используемого HTML-кода и файлов, на которые имеются ссылки.

Сгенерированные файлы JavaScript также оптимизируются для производственного использования, а также минифицируются на этапе компиляции.

Сервер Express

Одним словом, чтобы обслуживать приложение Angular с http-сервера, нам просто нужно предоставить файл `index.html`, который находится в каталоге `dist`. В идеале этот файл `index.html` должен находиться в базовом каталоге http-сервера, поэтому это будет файл по умолчанию, если ничего не указано. Давайте для этого настроим сервер Express.

Сперва мы создадим каталог Express и файл `main.ts` в этом каталоге, который будет содержать код нашего сервера Express. Файл `express/main.ts` выглядит так:

```
import express from 'express';
import path from 'path';

let app = express();

app.use(`/`, express.static(__dirname + '/angular-sample'));

app.get(`*`, (req: any, res: any) => {
  res.status(200).sendFile(
    path.join(__dirname + '/angular-sample/index.html'));
});

app.listen(9000, () => {
  console.log(`Express server listening on PORT: 9000`);
});
```

Здесь вначале идут обычные операторы `import` для модулей `express` и `path`, а затем мы создаем переменную `app`, которая является экземпляром `express`. После этого мы настраиваем Express, чтобы установить каталог по умолчанию как `angular-sample` с помощью функции `express.static`. Мы будем запускать сервер из каталога `dist`, поэтому этот оператор указывает, что сервер должен искать все статические файлы в подкаталоге `angular-sample`. Это означает, что любой HTML- или CSS-файл по умолчанию будет обслуживаться из подкаталога `angular-sample`.

Затем у нас есть один обработчик, который сообщает, что любой GET-запрос вернет файл `angularsample/index.html`. По сути, этот единственный обработчик будет перенаправлять все `GET`-запросы в наше приложение. Это означает, что, например, вызов `localhost:9000/test/test` фактически вернет файл `index.html`. Тогда Angular получает шанс интерпретировать URL-адрес `/test/test` и может инициировать поведение на основе собственных внутренних маршрутов. Мы по-прежнему можем указывать маршруты на нашем сервере Express, но любой URL-адрес, который не соответствует Express, будет автоматически передан в Angular. Мы рассмотрим маршруты в Angular чуть позже, но имейте в виду,

что Express получает первый шанс сопоставить маршрут, и если он не совпадает, то передается в Angular с помощью этой директивы.

Теперь наше приложение Express запускается и прослушивает порт `9000`.

Однако прежде чем мы сможем запустить сервер Express, нам нужно будет настроить TypeScript для компиляции файла main.ts и вывести результаты в каталог `dist`. Для этого нам потребуется создать файл `tsconfig.json` в каталоге `express`:

```
{
   "compilerOptions": {
      "target": "es5",
      "module": "commonjs",
      "lib": [
        "es2016",
        "dom"
      ],
      "outDir": "../dist",
      "strict": true,
      "moduleResolution": "node",
      "esModuleInterop": true
   }
}
```

Это стандартный файл `tsconfig.json`, который мы использовали ранее в Node. Единственное заметное изменение – это свойство `outDir`, которое будет помещать любые сгенерированные файлы JavaScript в каталог `../dist`. Это тот же каталог, который Angular использует для выходных данных. Мы можем скомпилировать наш сервер (из корневого каталога проекта) следующим образом:

```
tsc -p express
```

Опция `-p compile` позволяет нам указать подкаталог для компиляции. В данном случае это подкаталог с именем express. Следовательно, все файлы в каталоге `express` будут скомпилированы с использованием файла `tsconfig.json`, найденного в этом каталоге.

Обратите внимание, что теперь у нас есть два проекта TypeScript в одном корневом каталоге проекта. У нас есть проект Angular, исходный код которого находится в каталоге src, и проект Express, исходный код которого находится в каталоге `express`. Это означает, что они будут использовать один и тот же каталог `node_modules` при компиляции. Поэтому нам нужно будет установить соответствующие node-модули Express из базового каталога проекта:

```
npm install express
npm install @types/node -saveDev
```

Команда компиляции для подкаталога Express теперь сгенерирует файл `main.js` в каталоге `dist`. Мы можем запустить наш сервер:

```
node dist/main
```

Конфигурация сервера

Теперь, когда у нас есть работающий сервер Express, давайте проведем рефакторинг кода, чтобы сделать возможной простую настройку. Как было упомянуто ранее, конфигурация сервера Express может легко меняться в зависимости от того, где он развернут. В локальной среде разработки мы, возможно, захотим использовать локальный SMTP-сервер, но в тестовой и производственной средах нам потребуется настоящий. Наличие набора файлов `config` для управления этими настройками означает, что мы можем развернуть один и тот же код в нескольких средах без перекомпиляции кода.

Чтобы с легкостью провести синтаксический анализ файлов конфигурации, мы будем использовать пакет Node `config`, который можно установить так:

```
npm install config
npm install @types/config -saveDev
```

Теперь мы можем обновить наш файл main.ts:

```
import config from 'config';

...существующий код...

enum ConfigOptions {
   PORT = 'port'
}

let port = 9000;

if (config.has(ConfigOptions.PORT)) {
   port = config.get(ConfigOptions.PORT);
} else {
   console.log(`no port configuration found, using default :
      ${port}`);
}

app.listen(port, () => {
   console.log(`Express server listening on PORT: ${port}`);
});
```

Здесь мы импортировали модуль `config`, как обычно. Затем мы определяем строковое перечисление с именем `ConfigOptions`, которое будет содержать строковые константы, которые будут использоваться в качестве ключей для наших оп-

ций конфигурации. На данном этапе у нас есть только одна опция `PORT`, которая соответствует ключу порта.

После этого мы определяем локальную переменную port и устанавливаем для нее значение по умолчанию `9000`. Следующая строка запрашивает наши параметры конфигурации, чтобы увидеть, было ли определено значение с именем `port`, используя функцию `config.has`. Если у нашей опции конфигурации есть ключ port, мы перезаписываем локальную переменную port, используя это значение. В противном случае мы записываем сообщение в консоль, чтобы указать, что параметр конфигурации с именем `port` не был найден, и возвращаемся к значению по умолчанию `9000`.

Затем мы запускаем сервер Express на настроенном порту, указав локальную переменную port в вызове `app.listen`.

Имея этот код конфигурации, мы можем теперь создать каталог `config`, а внутри него файл с именем `default.json`:

```
{
   "port": "9999"
}
```

Пакет `config` попытается найти файл с именем `config/default.json` для использования в качестве исходного файла конфигурации. После этого, когда мы запустим наш сервер Express, он изменит номер порта на `9999`, как указано в файле конфигурации.

Преимущество использования пакета, такого как `config`, для настройки нашего сервера заключается в том, что мы также можем использовать переменные среды или переопределения командной строки для установки каждого параметра `config`. Это означает, что мы можем запустить нашу программу `main.js`:

```
node dist/main --NODE_CONFIG='{"port": "8888"}'
```

Здесь мы использовали командную строку, чтобы переопределить наш параметр конфигурации `port` из командной строки. Пожалуйста, обратитесь к документации по пакету `config` для получения дополнительной информации.

Ведение журнала сервера

В любой производственной системе журналы сервера являются неотъемлемой частью мониторинга и устранения неисправностей работающей системы. Для приложений Node это ничем не отличается. Есть несколько удобных модулей, которые мы кратко рассмотрим, чтобы улучшить процедуру ведения журналов на нашем сервере, так что это может быть более полезным в производственных условиях.

Первый модуль называется `rotating-file-stream` и будет вращать журналы на сервере в соответствии с настраиваемым набором правил. При входе в производственную среду мы должны следить за тем, чтобы журналы сервера не становились слишком большими, иначе мы рискуем исчерпать место на диске. При вращении журнала спустя какой-то период времени будет создан новый файл с новым именем, и можно настроить максимальный размер файлов журналов. Это гарантирует, что мы можем ограничить количество генерируемых файлов и не подвергнемся риску перегрузить сервер.

Вторая библиотека называется `moment-timezone` и используется для манипулирования датами и временем с учетом настроенного часового пояса. Поэтому мы можем представлять одну и ту же дату и время в нескольких часовых поясах. У нее также имеется полный набор утилит, которые могут обрабатывать любой исходный формат даты, что невероятно полезно при работе с датами и временем. Конечные точки REST обычно используют строковый формат ISO для обозначения дат, а они обычно находятся в формате UTC. JavaScript, однако, использует собственный объект Date, который обозначает количество миллисекунд с начала эпохи. Правильное преобразование между этими обозначениями дат может стать очень запутанным, обескураживающим и отнимать много времени. Займитесь несколькими часовыми поясами, и это может быстро превратиться в кошмар. Однако библиотека `moment-timezone` делает работу с датами и временем внутри часового пояса очень простой.

Мы обновим наш серверный код тремя способами. Во-первых, мы добавим параметр конфигурации `timezone`, как мы это делали ранее для порта сервера. Во-вторых, мы настроим вращающийся поток файлов для журналов нашего сервера. Наконец, мы напишем простую функцию ведения журнала, которая будет использовать вращающийся файловый поток и наш настроенный часовой пояс для записи сообщений в консоль как по UTC, так и по местному времени.

Прежде всего выполним конфигурацию нашего часового пояса:

```
enum ConfigOptions {
   PORT = 'port',
   TIMEZONE = 'timezone'
}

let timezone = "Australia/Perth";

if (config.has(ConfigOptions.TIMEZONE)) {
   timezone = config.get(ConfigOptions.TIMEZONE);
} else {
   serverLog(`no timezone specified, using ${timezone}`)
}
```

Здесь мы добавили значение перечисления `TIMEZONE` в `ConfigOptions` и установили внутреннее значение строки равным `'timezone'`. Затем мы создаем ло-

кальную переменную с именем `timezone` и устанавливаем для нее значение по умолчанию `'Australia/Perth'`. После этого мы проверяем через функцию `config.has`, есть ли у нашей текущей конфигурации времени выполнения значение переопределения для `timezone`, и используем его, если оно есть.

Это ничем не отличается от логики, которую мы использовали для установки свойства port по умолчанию. Мы обсудим функцию `serverLog` чуть позже.

Далее мы установим модуль `rotating-file-stream`:

```
npm install rotating-file-stream
```

Мы можем создать вращающийся журнал следующим образом:

```
var logDirectory = path.join(__dirname, 'log')

// Убеждаемся, что каталог журналов существует;
fs.existsSync(logDirectory) || fs.mkdirSync(logDirectory)

// Создаем вращающийся поток записи;
var accessLogStream = rfs('main.log', {
   interval: '10s', // rotate every 10 seconds
   path: logDirectory,
   maxFiles: 7
});
```

Здесь мы начинаем с определения имени каталога для файлов журнала на основе текущего рабочего каталога сервера Express с помощью функции `path.join` и переменной `__dirname`. Затем мы убеждаемся, что каталог существует, используя логический тип данных значение или сравнение. Если `fx.existsSync` возвращает значение `true`, мы не будем выполнять функцию `fs.mkdirSync`. Если он возвращает значение `false`, поскольку каталог не существует, будет выполнена функция `fs.mkdirSync`, которая создаст каталог.

Затем мы создаем вращающийся файловый поток с именем `accessLogStream`, вызывая функцию `rfs`, содержащую несколько параметров. Первый – это имя файла журнала, а второй параметр – это набор параметров конфигурации для этого файлового потока. Мы установили интервал вращения на `10` секунд, просто с целью тестирования, и установили путь к файлам журналов в нашем недавно созданном каталоге журналов. После этого мы указываем, что максимальное количество файлов, которое нам нужно в этом каталоге, равно 7.

Эти параметры можно легко настроить для производственного использования, и также имеет смысл извлечь эти значения из настройки конфигурации.

Получив вращающийся файловый поток, мы можем создать для нашего сервера функцию для записи сообщений:

```
function serverLog(message: string) {
   let now = moment.tz(timezone);
   let nowISOFormat = now.toISOString();
   let nowLocalFormat = now.format('YYYY-MM-DD HH:mm:ss.SSS');
   let logMessage = `UTC : ${nowISOFormat} :
      local : ${nowLocalFormat} : ${message}`;
   accessLogStream.write(`${logMessage}\n`);
   console.log(logMessage);
}
```

Здесь мы определили простую функцию с именем `serverLog`, которая принимает параметр `message` типа `string`. Первое, что мы делаем в этой функции, – это создаем экземпляр формата `moment date time` в текущем настроенном часовом поясе, вызывая функцию `moment.tz` и передавая строку часового пояса, которая в настоящее время установлена как `'Australia/Perth'`. Затем мы создаем переменную с именем `nowISOFormat`, вызывая функцию `toISOString` объекта `moment date time`. Помните, что в часовом поясе все даты хранятся в формате UTC, поэтому эта строка будет отображать дату в формате UTC в формате ISO. После этого мы создаем еще одну переменную с именем `nowLocalFormat`, вызывая функцию `format` для переменной now, и передаем формат даты. Этот вызов будет учитывать наш текущий настроенный часовой пояс, так как базовый моментный объект был создан с настройкой часового пояса.

Как только мы получили временные строки, мы записываем сообщение в консоль, показывающее как дату в формате UTC, так и местное время. Обратите внимание, что мы пишем сообщение в наш вращающийся файловый поток с помощью переменной `accessLogStream`, которую мы создали ранее. Мы также записываем сообщение в стандартную консоль.

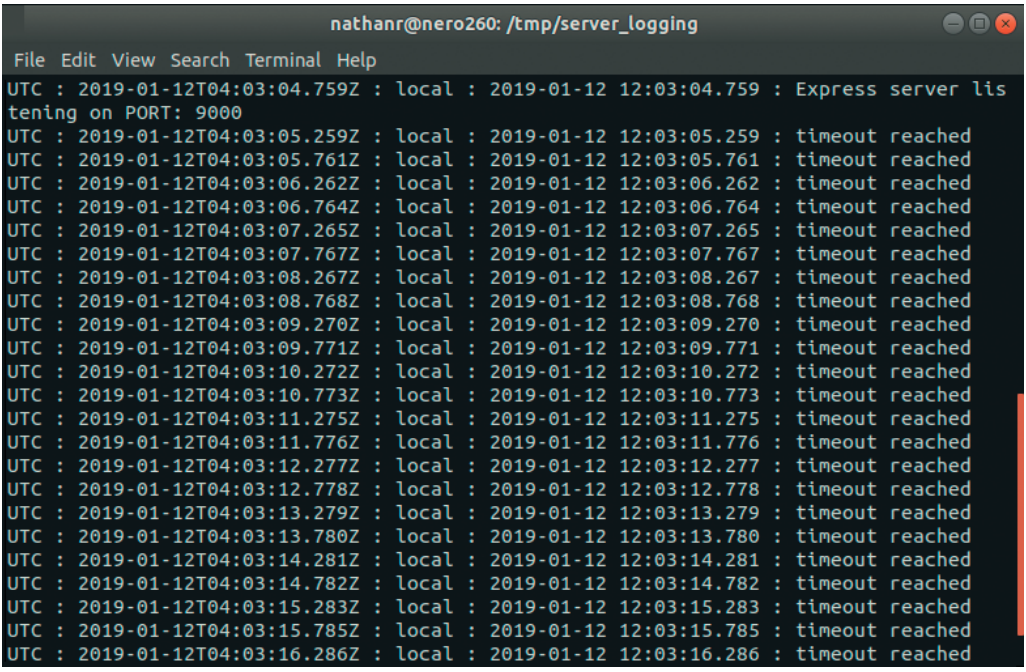
Чтобы проверить, что файловый поток работает правильно, давайте запишем сообщение, используя функцию `serverLog` каждые 500 миллисекунд:

```
setInterval(() => {
   serverLog(`timeout reached`);
}, 500);
```

Имея тестовую функцию, мы можем запустить наш сервер Express, как обычно:

```
node dist/main
```

Вывод на консоль теперь будет записывать сообщения каждые 500 миллисекунд, а мы в это же время будем вести запись в наш поток. Вывод теперь выглядит так:



Здесь видно, что функция `serverLog` записывает время в формате UTC, а также местное время и сообщение в консоль.

Мы увидели, что можно довольно легко настроить сервер Express с помощью пакета `config` и можно записывать сообщения во вращающийся поток файлов на нашем сервере. Мы также видели, как пакет `moment-timezone` помогает нам, предоставляя простые служебные функции для обработки объектов даты и времени в нескольких часовых поясах.

Опыт взаимодействия

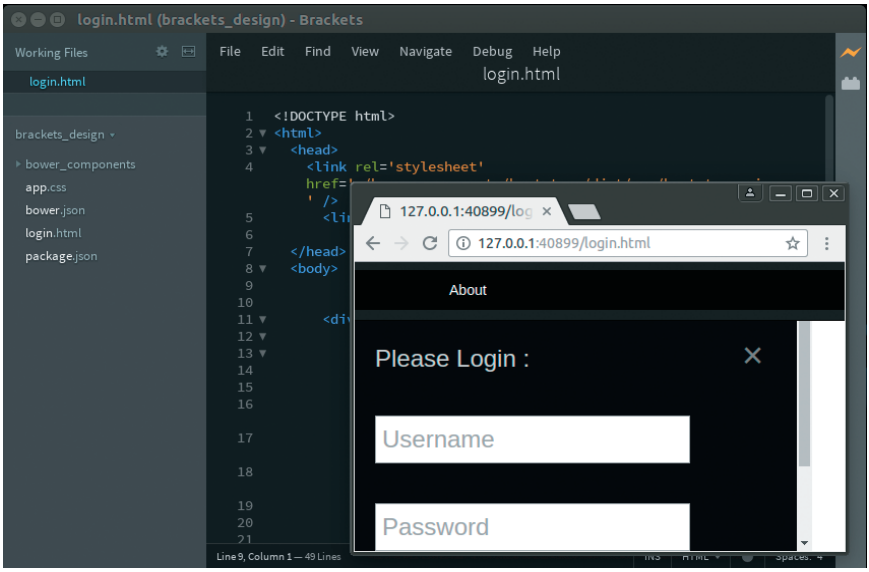
В начале каждого веб-проекта обсуждаются требования к **пользовательскому интерфейсу**. Как будет выглядеть приложение, какие CSS-стили оно будет использовать, и как наш пользователь будет взаимодействовать с системой? Пользовательский интерфейс – это простота в использовании, интуиция и простой рабочий процесс. Как таковой он может либо создать, либо испортить хороший сайт. Ориентация на хороший **опыт взаимодействия** (**UX**) означает, что многие компании нанимают команды специалистов для разработки пользовательского интерфейса для внешнего вида или для разработки опыта взаимодействия, включая рабочий процесс. В зависимости от навыков команды UX результат их деятельности может представлять собой набор изображений, которые показывают, как должен выглядеть опыт взаимодействия, или это может быть набор HTML-страниц и CSS-файлов.

Однако придет время, когда каждый разработчик должен будет собрать пользовательский интерфейс, поэтому понимание процесса и работа с инструментами, разработанными для проектирования, являются необходимым шагом при создании приложений. В этом разделе мы будем работать с Bootstrap в качестве основы для стилевого оформления нашего веб-сайта, а также познакомимся с редактором Brackets, который отлично подходит для быстрого прототипирования сайтов.

Использование Brackets

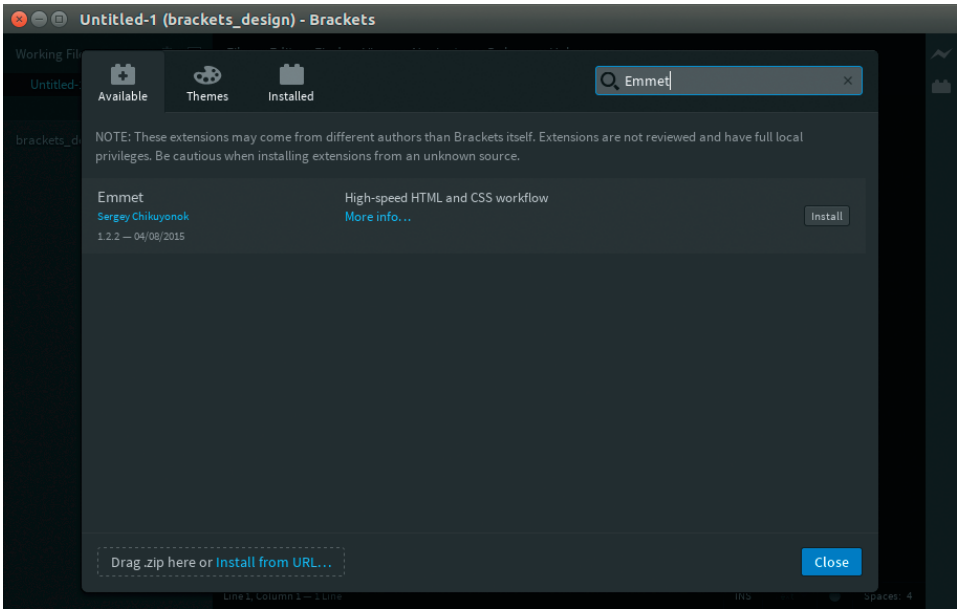
Работая с HTML-кодом и CSS-стилями на этапе проектирования, мы постоянно редактируем и настраиваем как HTML-файлы, так и таблицу CSS-стилей, чтобы наши страницы выглядели хорошо. Одним из лучших инструментов для этой работы является редактор `Brackets`. Brackets – это редактор с открытым исходным кодом, специально предназначенный для веб-дизайнеров и frontend-разработчиков. Он обладает множеством функций, предназначенных для быстрого редактирования HTML- и CSS-элементов, включая предварительный просмотр в режиме реального времени, щечок правой кнопкой мыши, чтобы редактировать CSS-стили, палитры цветов и многое другое. Однако одной из самых удобных функций является предварительный просмотр в режиме реального времени.

В режиме предварительного просмотра открывается отдельное окно браузера, и любые изменения, внесенные в ваши HTML- или CSS-файлы, будут автоматически обновляться в браузере. Мгновенная обратная связь при применении CSS-стилей или редактировании HTML-кода невероятно экономит время. Brackets с окном предварительного просмотра в режиме реального времени показан на приведенном ниже скриншоте:



Установить Brackets так же просто, как загрузить установщик с сайта `brackets.io` и запустить его. После установки мы можем улучшить функциональность по умолчанию с помощью расширений. У Brackets очень удобный и простой менеджер расширений, который помогает найти и установить доступные расширения. Brackets также автоматически уведомит нас, когда будут доступны обновления для этих расширений.

Чтобы установить расширение, нажмите **File | Extension Manager** (**Файл | Менеджер расширений**) или щелкните значок кирпичика LEGO на вертикальной боковой панели справа. Мы будем использовать одно расширение под названием Emmet, но в Brackets доступны буквально сотни расширений. В строке поиска введите Emmet, а затем нажмите кнопку **Install** (автор: *Сергей Чикуёнок*):



В Brackets нет концепции проекта *как такового*, вместо этого он просто работает с корневой папкой. Давайте создадим новую папку в нашей файловой системе, а затем откроем ее в Brackets, используя **File | Open Folder** (**Файл | Открыть папку**).

Использование Emmet

Давайте теперь создадим простую HTML-страницу с помощью **File | New** (**Файл | Новый**) или просто нажав сочетание клавиш *Ctrl+N*.

Вместо того чтобы писать наш HTML-файл вручную, мы будем использовать Emmet для генерации HTML-кода. Введите следующую строку:

```
html>head+body>h3{index.html}+div#content
```

Теперь нажмите *Ctrl+Alt+Enter* или в меню **File** выберите **Emmet** | **Expand Abbreviation** (**Emmet** | **Развернуть аббревиатуру**).

Вуаля! Emmet сгенерировал следующий HTML-код за миллисекунду. Неплохо для одной строки кода:

```
<html>
   <head></head>
   <body>
      <h3>index.html</h3>
      <div id="content"></div>
   </body>
</html>
```

Нажмите сочетание клавиш *Ctrl+S*, чтобы сохранить файл, и введите `index.html` в качестве имени файла.

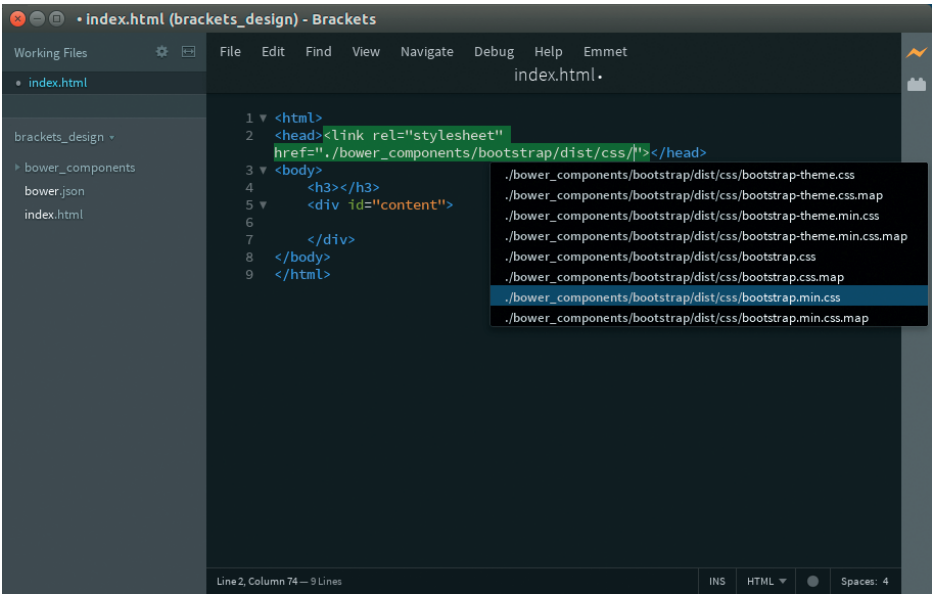
Только после того, как мы сохраним файл, Brackets начнет выполнять подсветку синтаксиса на основе расширения файла.

Давайте внимательнее посмотрим на строку аббревиатуры Emmet, которую мы ввели ранее. Emmet использует символ >, чтобы создать дочерний элемент, и символ + для обозначения элемента одного уровня. Если мы используем фигурные скобки {} рядом с элементом, то это означает, что для содержимого данного элемента будет установлено значение, указанное внутри скобок. Итак, строка Emmet, которую мы ввели ранее, гласит: создайте HTML-тег с дочерним тегом `head`, а затем создайте еще один дочерний HTML-тег с именем `body`. В теге `body` создайте тег `h3` с содержимым `index.html`, а затем создайте `div`-тег с идентификатором `content`. Перейдите на сайт Emmet (`emmet.io`) для получения дополнительной информации и не забывайте держать под рукой шпаргалку (`docs.emmet.io.cheatsheet`), когда вы изучаете сочетания клавиш Emmet и работаете с ними.

Давайте теперь добавим тег сценария в наш файл `index.html`. Переместите курсор между тегами `<head> </head>` и введите следующую строку:

```
link
```

Теперь нажмите сочетание клавиш *Ctrl+Alt+Enter,* чтобы Emmet сгенерировал полный тег `<link>`, и удобно поместите наш курсор между кавычками, установленными для имени файла. Имя файла, которое мы ищем, – `bootstrap-min.css`. Продолжайте и начните с ввода `./`. Обратите внимание, что Brackets понимает, что вы ищете CSS-файл, и включит Intellisense или опции автодополнения кода, чтобы помочь вам найти этот файл:



Подключив файл `bootstrap.min.css`, мы можем приступить к конкретизации содержимого нашей HMTL-страницы. Прежде чем продолжить редактирование этого HTML-файла, перейдите к окну предварительного просмотра в режиме реального времени, нажав значок молнии в правой части окна. Когда мы будем вносить изменения в HTML-файл, они будут автоматически отражаться в окне предварительного просмотра.

В начале тега `<body>` мы можем создать контейнер Bootstrap с помощью следующей строки:

```
div.container.h-100>div.row>div.col
```

В результате этого будет сгенерирован следующий HTML-код:

```
<div class="container h-100">
   <div class="row">
      <div class="col"></div>
   </div>
</div>
```

Эта строка сообщает следующее: создайте `div`-элемент с классом `container` и `h-100`. Внутри этого элемента создайте дочерний `div`-элемент с классом `row`, а внутри него создайте еще один элемент с классом `col`. Результатами обработки этой строки является HTML-код, как видно из предыдущего примера.

Теперь мы можем поместить наш курсор внутрь самого внутреннего тега `<div>` и создать компонент `jumbotron` с помощью следующей строки:

```
div.jumbotron>img+h2{Jumbotron}+div.form-group
```

В результате этого будет создан div-элемент с CSS-классом jumbotron, а затем дочерний элемент img, элемент h2 и div-элемент:

```
<div class="jumbotron">
   <img src="" alt="">
   <h2>Jumbotron</h2>
   <div class="form-group"></div>
</div>
```

Создание панели входа

Сгенерированный нами HTML-код формирует основу центрированной панели, которую мы можем использовать в качестве панели входа в систему. Мы бы хотели, чтобы jumbotron располагался по центру страницы, как по горизонтали, так и по вертикали. Давайте применим стили Bootstrap и CSS для достижения этой цели.

Пока что у нас есть внешний элемент с CSS-классом 'container' и внутренний элемент с классом 'jumbotron'. Давайте обновим внешний элемент:

```
<div class="container h-100">
   <div class="row align-items-center h-100">
      <div class="col-6 mx-auto">
         <div class="jumbotron">
            ... jumbotron body ...
         </div>
      </div>
   </div>
</div>
```

Здесь мы показываем внешний элемент с CSS-классом 'container', в котором также есть класс 'h-100'. Это означает, что он будет занимать 100 % высоты экрана. Далее у нас идет -элемент с классами 'row align-items-center h-100'. Эти стили означают, что весь ряд будет отцентрирован с высотой 100 %. Внутри этого элемента мы создаем -элемент со стилем 'col-6 mx-auto'. По сути, это колонка размером 6, что означает, что она будет занимать половину экрана. Когда ряд будет центрирован, эта колонка также будет центрирована внутри ряда. Теперь мы можем обновить -элемент классом 'jumbotron':

```
<div class="jumbotron text-center shadow-lg p-3 mb-5 bg-white
rounded">
   <img src="logo.jpg" class="img-fluid">
   <h2>Please Login</h2>
   <div class="form-group">
      <input class="form-control" type="text"
```

```
          placeholder="Username">
        </div>
        <div class="form-group">
          <input class="form-control" type="password"
          placeholder="Password">
        </div>
        <div class="form-group">
          <button class="btn btn-primary" type="submit" >Login
          </button>
          <button class="btn btn-primary">Login with Google</button>
        </div>
        <div *ngIf="error">
          <div class="alert alert-danger">
            {{error}}
        </div>
      </div>
    </div>
```

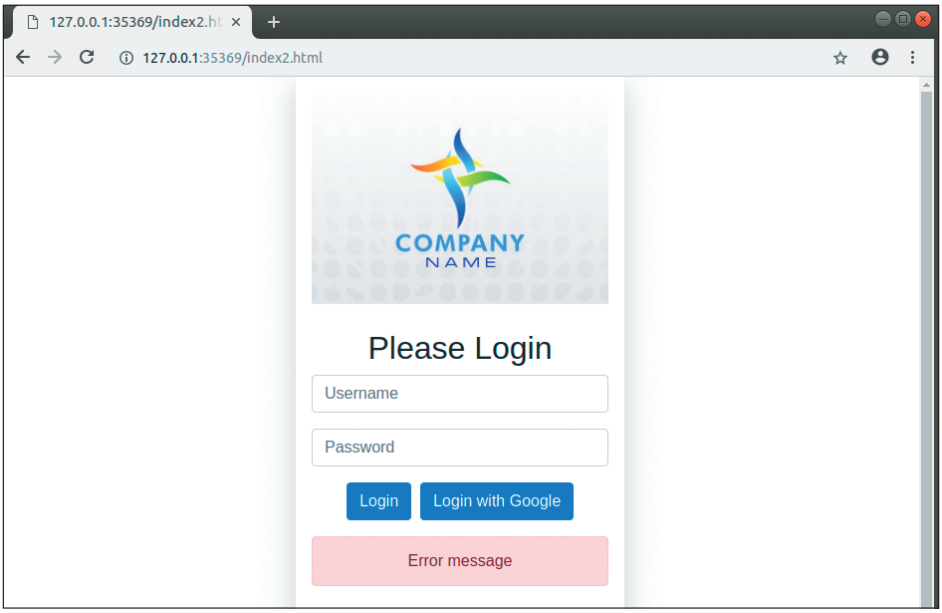
Здесь мы определили полный HTML-код для внутреннего элемента `jumbotron`. У него есть ряд элементов:

- логотип;
- элемент `<input>` для ввода имени пользователя;
- элемент `<input>` для ввода пароля;
- элемент `<button>` с текстом Login;
- элемент `<button>` с текстом Login with Google;
- элемент для отображения сообщения об ошибке.

Помимо этого HTML-кода, нам нужен только один CSS-стиль (определенный в `app.css`):

```
body,html {
  height:100%;
}
```

Данный код в сочетании с CSS будет выглядеть на экране так:



Здесь видно, что у нас есть центрированный jumbotron, который содержит форму входа. У этой формы два элемента ввода: **имя пользователя** и **пароль**, а также кнопки **Login** и **Login with Google**. Под этими кнопками находится область вывода сообщений об ошибках.

Имея этот код, мы можем использовать скобки, чтобы быстро и легко создавать новые стили, или адаптировать этот макет по своему усмотрению. Мы потратили немного времени или усилий на создание этих макетов; мы просто настраивали HTML и CSS. Эту фазу проектирования проекта поэтому можно выполнить очень быстро, используя только минимальные знания HTML и CSS. С помощью таких шаблонов мы также можем начать беседу с клиентами, чтобы определить, соответствует ли внешний вид сайта ожидаемому, без необходимости создания целого приложения.

Теперь, когда у нас есть первоначальный дизайн, мы можем приступить к реализации этих экранов в нашем приложении Angular.

Аутентификация

Почти каждый сайт, который мы посещаем, требует регистрации и входа в систему. Несмотря на то что для просмотра содержимого может и не потребоваться вход в систему, как только мы начнем взаимодействовать с сайтом и добавим элементы в корзину, например, нам нужно создать какой-нибудь профиль и выполнить вход. Корпоративные веб-сайты обычно запрашивают логин, прежде чем мы

сможем даже начать взаимодействовать с сайтом, поскольку права доступа на то, что вы можете делать на сайте, зависят от вашей роли.

К счастью, мы можем получить доступ к широкому спектру сайтов, используя процесс внешней аутентификации, например с помощью Facebook или Google. Это означает, что мы можем повторно использовать логин Google или Facebook, и нам не нужно запоминать отдельную комбинацию, состоящую из имени пользователя и пароля для каждого сайта.

В этом разделе мы обновим наше приложение, чтобы показать, как работает аутентификация в Angular, и реализуем как процесс входа в систему с именем пользователя/паролем, так и процесс внешней аутентификации. Попутно мы исследуем маршрутизацию в Angular, используя средства защиты для блокировки доступа к определенным областям сайта, а затем используем JWT-токены вместо файлов cookie для хранения пользовательской информации. Наконец, мы покажем, как интегрировать аутентификацию Google в качестве примера интеграции внешней аутентификации в наш сайт.

Маршрутизация в Angular

В `главе 10` *«Модуляризация»* в разделе *«Использование Express с Node»* мы рассмотрели, как сервер Express использует маршрутизацию для создания разных страниц на основе URL-адреса, введенного в браузер. Мы также показали, как можно заставить Express перенаправить браузер на другую страницу в зависимости от того, совершил пользователь вход или нет. Наше приложение, однако, является **одностраничным**, поэтому нам необходимо реализовать любую навигацию по страницам на основе URL-адреса внутри самого Angular. Для этого мы будем использовать маршрутизацию.

Маршрутизация в Angular аналогична маршрутизации в Express, где мы определяем, какие маршруты нам нужны, а затем присоединяем класс для обработки этого маршрута. Чтобы настроить маршрутизацию в Angular, давайте создадим файл с именем `app.routing.module.ts` в каталоге `app`:

```
import { NgModule } from "@angular/core";
import { RouterModule, Routes } from '@angular/router';

export const routes: Routes = [
   { path: 'login', component: LoginPanelComponent },
   { path: '', component: SecureComponent },
];
@NgModule({
   imports: [RouterModule.forRoot(routes)],
   exports: [RouterModule]
})

export class AppRoutingModule { }
```

Здесь мы, как обычно, импортировали NgModule из @angular/core, а затем импортировали RouterModule и Routes из @angular/router. Затем мы создаем постоянную переменную route, которая представляет собой массив. Первый элемент этого массива содержит свойство path со значением 'login' и свойство компонента LoginPanelComponent.

Второй элемент этого массива содержит пустое свойство path и свойство компонента SecureComponent. Такова суть маршрутизации в Angular.

Переменная routs объявляет, что если мы добавим значение 'login' в наш URL-адрес, то компонентом, который будет обрабатывать этот маршрут, будет LoginPanelComponent. Таким образом, в нашей среде разработки URL-адрес http://localhost:4200/login будет преобразован в компонент LoginPanelComponent. Любой другой адрес будет преобразован в SecureComponent.

Последняя часть этого файла – создание класса с именем AppRoutingModule и его декорирование с помощью декоратора @NgModule, у которого есть свойства import и export. Свойство import используется для регистрации переменной определенных маршрутов в RouterModule, а свойство export экспортирует RouterModule. Это похоже на то, как мы регистрировали маршруты в Express, где нам нужно было выполнить повторный экспорт измененного маршрута после внесения изменений в конфигурацию.

Если мы попытаемся осуществить компиляцию на данном этапе, появятся сообщения об ошибках, потому что ни один из этих компонентов еще не существует. Давайте создадим эти два компонента, используя интерфейс командной строки Angular:

```
ng generate component login-panel --skip-import
```

С помощью этой команды мы сгенерирум каталог с именем login-panel в каталоге src и создадим необходимые CSS-, HTML- и .ts- файлы для нового компонента. Мы используем --skipimport, чтобы пропустить шаг, который пытается включить этот компонент в ближайший модуль. Поэтому нам самим нужно будет вручную включить этот компонент в наш файл app.module.ts. Мы можем создать SecureComponent аналогичным образом:

```
ng generate component layout/secure --skip-import
```

С помощью этой команды будет создан компонент secure.component.ts в каталоге src/app/layout/secure. Мы пока оставим сгенерированный HTML-код по умолчанию, чтобы убедиться, что наша маршрутизация в Angular работает правильно. Имея эти компоненты, мы можем получить файл app.routing.module.ts для правильной компиляции, импортировав компоненты:

```
import { LoginPanelComponent } from "./login-panel/login-panel.
  component";
```

```
import { SecureComponent } from "./layout/secure/secure.
  component";
```

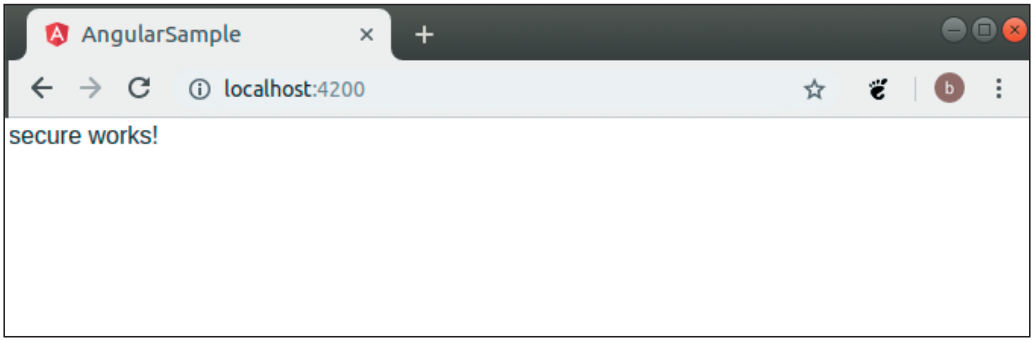
Как только класс `AppRoutingModule` будет компилироваться правильно, мы можем включить эти компоненты в наш файл app.module.ts:

```
@NgModule({
   declarations: [
      ...имеющиеся компоненты...
      SecureComponent,
      LoginPanelComponent
   ],
   imports: [
      BrowserModule,
      AppRoutingModule
   ],
   providers: [],
   bootstrap: [AppComponent]
})
export class AppModule { }
```

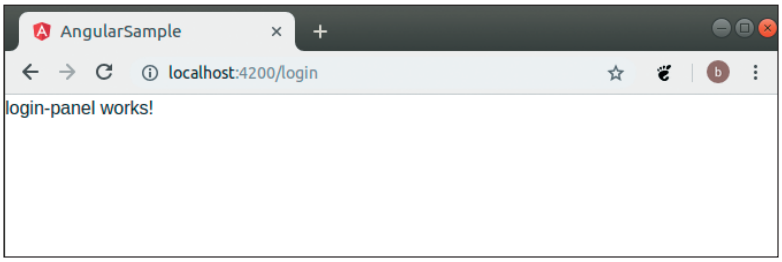
Последний шаг в настройке маршрутизации – создание экземпляра и активация маршрутизатора при первой визуализации HTML-файла. Это достигается путем замены HTML-кода в файле `app.component.html`:

```
<router-outlet></router-outlet>
```

Если мы скомпилируем и запустим наше приложение сейчас, то будем автоматически перенаправлены на HTML-файл SecureComponent, так как он регистрируется для визуализации, когда не указан дочерний URL-адрес:



Если мы теперь изменим URL-адрес на `localhost:4200/login`, маршрутизатор Angular перенаправит наш запрос на обработку `LoginPanelComponent` и, следовательно, отобразит следующее:



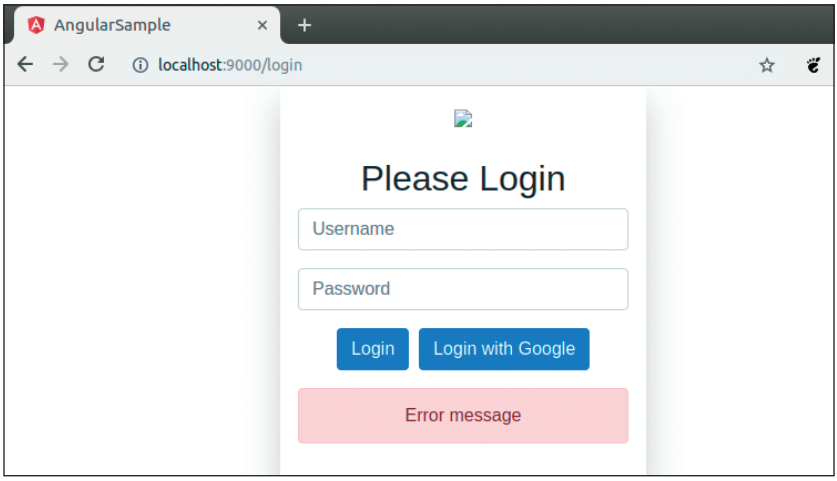
Теперь у нас есть маршрутизация, и мы можем отобразить два разных компонента на основе URL-адреса браузера.

Однако это означает, что нам потребуется переместить существующий HTML-, CSS-код и код в `.ts`, который мы использовали в `AppComponent`, в новый `SecureComponent`. Для краткости мы не будем обсуждать детали того, как это делается, но скопировать HTML-код, CSS-стили и функции, реализованные для `AppComponent`, в новый `SecureComponent` и правильно скомпилировать его не представляет труда.

Использование HTML-кода, предназначенного для пользовательского интерфейса

В разделе *«Опыт взаимодействия»* мы показали, как использовать Brackets для быстрой модификации HTML- и CSS-кода, чтобы достичь желаемого дизайна страницы за очень короткий промежуток времени. К счастью, интегрировать этот HTML-код в наше приложение проще простого.

Если мы просто скопируем и вставим содержимое тега `<body>` в наш файл `loginpanel.component.html`, мы почти у цели:



Как видно, LoginPanelComponent отображает наш HTML-код, спроектированный в Brackets, за исключением логотипа. Чтобы исправить это, мы должны иметь возможность подключать изображения как часть нашего шага компиляции в Angular, чтобы они содержались в выходном каталоге dist.

Ключ к механизму, который использует Angular, можно найти в файле angular.json в дереве свойств projects | angular-sample | architect | build | options. Он носит название "assets":

```
"assets": [
  "src/favicon.ico",
  "src/assets"
],
```

Здесь видно, что в Angular существует концепция "assets", которая будет включена в сборку. Она состоит из src/favicon.ico и всего, что находится в каталоге с именем src/assets. Это означает, что мы можем создать каталог assets в каталоге src и скопировать изображение нашего логотипа в этот каталог. Это приведет к тому, что каталог ресурсов будет включен в наш каталог dist, в папку angular-sample/assets. Помните, что наше приложение в Express обслуживает любые статические файлы, которые оно находит в каталоге angular-sample, с помощью следующего параметра конфигурации:

```
app.use (`/`, express.static (__dirname + '/ angular-sample'));
```

До этого момента мы использовали данный параметр конфигурации для обслуживания базового файла index.html в Angular, но это также будет работать для любого подкаталога, найденного в каталоге angular-sample. Это означает, что мы можем скопировать любые изображения, которые нам нужны для нашего приложения, в папку assets и можем изменить свойство src тега <img> следующим образом:

```
<img src = "/assets/logo.jpg" class = "img-Fluid">
```

Здесь наш сгенерированный LoginPanelComponent обращается к изображению логотипа в статическом каталоге с именем assets, как показано в свойстве src, которое теперь носит название assets/logo.jpg. Таким образом, сгенерированный HTML-код будет правильно отображать логотип.

Стражи аутентификации

Теперь наше приложение может отображать SecureComponent или LoginPanelComponent в зависимости от URL-адреса, предоставленного браузером. В настоящее время, если URL-адрес не указан, приложение отобразит SecureComponent, который является основным экраном нашего сайта. Но что, если нам нужно убедиться, что SecureComponent отображается только в том

случае, если пользователь уже зашел на сайт? Другими словами, как защитить пользователей от доступа к разделам нашего сайта, если они не совершили вход? Angular использует для этой цели стражей аутентификации или Auth Guards.

Основная предпосылка Auth Guard – позволить нашему коду проверить, имеет ли пользователь правильные полномочия для доступа к определенному маршруту. Это делается путем создания функции, которая возвращает значение `true`, если пользователю разрешен доступ, или `false`, если это не так. Мы можем настроить Auth Guard в нашей конфигурации маршрутизации так:

```
export const routes: Routes = [
   {path: 'login', component: LoginPanelComponent},
   {path: '', component: SecureComponent,
      canActivate: [AuthGuard]},
];
```

Здесь мы обновили файл `app.routing.module.ts` и изменили конфигурацию маршрутизации для `SecureComponent`, добавив другое свойство с именем `canActivate`, которое обращается к классу `AuthGuard`. Этот класс должен реализовать единственную функцию с именем `canActivate`, которая должна возвращать логическое значение `true` либо `false`. Давайте создадим новый каталог с именем `src/guards` и в нем создадим новый файл `auth.guard.ts`:

```
import {Injectable} from '@angular/core';
import {CanActivate, ActivatedRouteSnapshot,
   RouterStateSnapshot, Router } from '@angular/router';

@Injectable({
   providedIn: 'root'
})
export class AuthGuard implements CanActivate {

constructor() { }
   canActivate(route: ActivatedRouteSnapshot, state:
      RouterStateSnapshot): boolean {
      return false;
   }
}
```

Мы создали класс с именем AuthGuard, который реализует интерфейс `CanActivate` и поэтому должен определить функцию `canActivate`. Мы не будем использовать аргументы `route` или `state` внутри этой функции, а просто вернем значение `false`. Это указывает на то, что никто пока не сможет получить доступ к нашему `SecureComponent`. Однако если мы запустим наше приложение сейчас, то получим один пустой экран.

Это происходит потому, что AuthGuard блокирует доступ к `SecureComponent`, поэтому отображается пустой экран, как и ожидалось, но приложение не было

настроено на что-либо еще. Что нам нужно сделать, так это перенаправить браузер на экран входа в систему, если доступ к SecureComponent был заблокирован. Мы можем сделать это очень легко с помощью внутренних возможностей маршрутизации Angular. Нам нужно будет изменить AuthGuard в двух местах:

```
export class AuthGuard implements CanActivate {
   constructor(private router: Router) { }

   canActivate(
      route: ActivatedRouteSnapshot,
      state: RouterStateSnapshot): boolean
      {
         this.router.navigate([`/login`]);
         return false;
      }
}
```

Здесь мы обновили функцию contructor для использования фреймворка внедрения зависимости Angular, чтобы получить доступ к экземпляру Router. В нашей функции canActivate мы использовали переменную private router, чтобы перенаправить браузер на URL-адрес входа, вызвав функцию navigate. Это гарантирует, что любой неаутентифицированный пользователь нашего приложения будет перенаправлен на компонент LoginPanelComponent.

Связывание формы входа

Теперь, когда у нас есть страж аутентификации и маршрутизация, работающая в нашем приложении, давайте обновим LoginPanelComponent, чтобы захватить имя пользователя и пароль, введенные в форму, и отправить их обратно на наш сервер Express. Для простоты мы будем использовать стандартное связывание данных на основе форм. Наш обновленный HTML-код выглядит так:

```
<div class="form-group">
   <input class="form-control" type="text" placeholder="Username"
      name="username" [(ngModel)]="username">
</div>

<div class="form-group">
   <input class="form-control" type="password"
      placeholder="Password" name="password"
      (ngModel)]="password">
</div>

<div class="form-group">
   <button class="btn btn-primary" type="submit"
      (click)="onLoginClicked()">Login</button>
```

```
    <button class="btn btn-primary" >Login with Google</button>
</div>

<div *ngIf="error">
    <div class="alert alert-danger">
        {{error}}
    </div>
</div>
```

Здесь мы использовали синтаксис `[(ngModel)]`, чтобы связать элемент управления вводом имени пользователя со свойством `username`. Мы также связали управление вводом пароля со свойством `password`. Затем мы установили действие (`click`) для кнопки *Login*, чтобы вызвать функцию `onLoginClicked`, когда пользователь нажимает эту кнопку. Наконец, мы привязали свойство `error` к панели ошибок в нижней части экрана. Мы используем директиву `*ngIf`, чтобы эта панель отображалась только в случае фактической ошибки.

Вот как выглядят изменения в файле `login-panel.component.ts`:

```
export class LoginPanelComponent implements OnInit {
    username: string;
    password: string;
    error: string;

    constructor(private router: Router) { }
        ngOnInit() { }

    onLoginClicked() {
        console.log(`LoginPanelComponent :
            this.username : ${this.username}`);
        this.error = "login not implemented yet.";
    }
}
```

Мы добавили к нашему компоненту свойства `username`, `password` и `error`, чтобы иметь возможность связать эти переменные с нашей формой. Мы обновили нашу функцию-конструктор для запроса объекта `Router`, чтобы мы могли перенаправить пользователя к нашему `SecureComponent` после успешного входа. Функция `onLoginClicked` просто записывает сообщение в консоль, чтобы убедиться, что связывание данных работает правильно. Наконец, мы устанавливаем свойство `error`, чтобы сообщить, что эта функция еще не реализована.

Прежде чем запускать приложение, нужно не забыть включить `FormsModule` в файл `app.module.ts`, в противном случае мы получим странные ошибки. Свойство `import` в `AppModule` теперь должно выглядеть так:

```
imports: [
    BrowserModule,
```

```
    AppRoutingModule,
    FormsModule
  ],
```

После внесения этих изменений у нас есть функциональная страница входа в приложение, которая принимает пользовательский ввод в элементах ввода имени пользователя и пароля и готова к использованию в функции `onLoginClicked`. Теперь мы готовы отправить нашу комбинацию имени пользователя и пароля на backend-сервер, используя метод `POST` для запроса аутентификации.

Использование HttpClient

Чтобы наш сервер Express получил POST, нам нужно создать обработчик маршрута и зарегистрировать его на нашем сервере. Давайте создадим файл с именем `express/route/userRoutes.ts`:

```
import * as express from 'express';
import { serverLog } from '../main';

var router = express.Router();

router.post(`/login`, (req: any, res: any, next: any) => {
   serverLog(`POST /login`);
   if (req.body.username && req.body.username.length > 0
      && req.body.password && req.body.password.length > 0) {
      res.json({ success: true });
   } else {
      serverLog(`/login - Error : Invalid username or password`);
      res.status(401).send('Invalid username or password');
   }
});

export { router };
```

Здесь мы создали обработчик маршрута для обработки события POST по URL-адресу `/login`. В этом обработчике мы записываем сообщение в консоль, используя функцию `serverLog`, а затем проверяем, что мы получили имя пользователя и пароль как часть тела запроса. Мы также проверяем, что строки `username` и `password` имеют длину `> 0`. Если они обе действительны, мы просто возвращаем строку в формате JSON с единственным свойством `success`, для которого установлено значение `true`. Если мы встречаем недопустимую комбинацию имени пользователя и пароля, мы возвращаем статус **401 HTTP** с сообщением.

Этот обработчик маршрута можно легко зарегистрировать на нашем сервере Express:

```
app.use ('/', userRoutes.router);
```

Наш сервер теперь может обрабатывать событие POST для конечной точки входа в систему и просто гарантирует, что имя пользователя и пароль не являются пустыми, и возвращает успех. Имея этот обработчик маршрута, мы можем обратить внимание на обновление нашего приложения Angular для использования этой конечной точки.

При интеграции с конечными точками REST не рекомендуется, чтобы мы взаимодействовали с этими конечными точками непосредственно из наших форм, а проходили через службу Angular. Создавая службы, мы придерживаемся принципа единственной ответственности, а также проектируем свой код так, чтобы он был более модульным. Наша форма просто отвечает за взаимодействие с пользователем, проверку ввода и обновление пользовательского интерфейса соответственно. Мы будем использовать службу Angular для отправки данных на наш сервер Express.

Давайте создадим сервис с помощью интерфейса командной строки Angular:

```
ng generate service services/user
```

Эта команда создаст обобщенную службу в файле src/app/services/user.service.ts. Мы обновим этот файл следующим образом:

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpHeaders } from '@angular/common/http';

@Injectable({
  providedIn: 'root'
})

export class UserService {
  constructor(private httpClient: HttpClient) { }

  authenticateUser(username: string, password: string)
    : Observable<Object> {
    const headers = new HttpHeaders();
    headers.append('Content-Type', 'application/json');
    const user = {
      username: username,
      password: password
    };
    return this.httpClient.post(
      '/login', user,
      { 'headers': headers });
  }
}
```

Здесь у нас есть простой класс обслуживания с именем UserService. Внутри конструктора мы запрашиваем экземпляр класса HttpClient через фрейм-

ворк внедрения зависимостей Angular. Затем идет функция `authenticateUser`, которая принимает два параметра, `username` и `password`. В этой функции мы устанавливаем экземпляр класса `HttpHeaders` и используем его для установки `Content-Type` в `application/json`. После этого мы создаем простой объект JavaScript, содержащий имя пользователя и пароль, а затем вызываем метод `post` экземпляра `httpClient`. Этот метод вызывается с тремя аргументами. Первый аргумент – URL-адрес конечной точки, на которую мы публикуем сообщение, в данном случае это `/login`. Второй аргумент – наш JavaScript-объект, а третий – наши HTTP-заголовки.

Как видно из этого кода, использовать объект `HttpClient` очень просто. Мы можем использовать любую из доступных функций: `get`, `post`, `put` или `delete`, – которые он предоставляет, для простой интеграции REST.

Интересной особенностью класса `UserService` является тип, возвращаемый из функции `authenticateUser`. Обратите внимание, что мы возвращаем результат вызова `httpClient.post`, который на самом деле является объектом `Observable`.

Использование Observable

Класс `HttpClient` в Angular использует Observables как механизм для обработки асинхронных вызовов к конечным точкам REST. Observables являются реализацией шаблона проектирования Observable и предоставляются JavaScript-библиотекой *Reactive Extensions for JavaScript* или просто RxJS. Шаблон проектирования Observable использует две концепции: регистрацию и уведомление. Объект, который заинтересован в событии, зарегистрируется в объекте Observable. Когда это событие происходит, объект Observable уведомит все объекты, которые зарегистрировались для этого события, с помощью шага уведомления. Библиотека RxJS расширила базовый шаблон проектирования Observable, чтобы обеспечить более детальный контроль над тем, когда и где возникают эти события. Однако преимущество использования объектов Observable заключается в том, что библиотека RxJS позволяет работать с так называемыми последовательностями Observable.

В качестве примера того, что такое последовательность Observable и что она может делать, давайте предположим, что мы читаем значения из потока событий, который будет посылать новое случайное число каждую секунду в течение 10 секунд. Это наша последовательность Observable. Мы будем получать событие один раз каждые 10 секунд с новым значением случайного числа.

Библиотека RxJS позволяет настраивать Observables для работы с этой последовательностью данных различными способами. Например, мы можем зарегистрировать Observable, который будет ждать, пока не произойдет определенное количество событий, прежде чем что-то сделать. Или мы можем подождать, пока не будут возвращены все данные, прежде чем сделать что-то еще. Мы также можем смешивать и сопоставлять Observables для выполнения таких задач, как вычис-

ление среднего значения последних трех чисел или запуск нового события, если число превышает определенный порог. Таким образом, преимущество использования Observables состоит в том, что мы можем реагировать не только на события, но и на последовательности событий.

Поэтому объекты Observable идеально подходят для асинхронного программирования и позволяют использовать очень простой синтаксис для регистрации события, а затем выполнить некоторый код, как только это событие было инициировано. Команда Angular внедрила RxJS в основную библиотеку Angular и широко использует ее для HTTP-запросов. Мы обсудим несколько интересных способов использования Observables и их объединения в следующем разделе.

Объекты Observables в RxJS используют функцию subscribe для регистрации события. Синтаксис этой функции выглядит так:

```
let testObservable: Observable<object>;
testObservable.subscribe(
   (success: Object) => {
      // успешный вызов
   },
   (error: any) => {
      // состояние ошибки
   },
   () => {
      // завершено или finally()
   }
)
```

Здесь мы вызываем метод `subscribe` с тремя функциями в качестве аргументов. Первая функция будет вызвана, когда Observable будет успешно возвращен. Вторая функция будет вызвана, если произошла какая-либо ошибка, а третья функция будет вызвана, когда Observable завершен. Обратите внимание, что функция завершения будет вызываться после функций успеха или ошибок, в зависимости от того, что произойдет. Это основной синтаксис работы с Observables.

Обратите внимание, что функции `error` и `complete` являются необязательными, поэтому их можно исключить из сигнатуры функции. Однако в производственных системах рекомендуется всегда подключать хотя бы обработчик ошибок.

Теперь давайте интегрируем `UserService` с `LoginPanelComponent`, чтобы иметь возможность отправить имя пользователя и пароль в конечную точку входа Express. Наши обновления для `LoginPanelComponent` выглядят так:

```
constructor(private userService: UserService,
private router: Router) { }

onLoginClicked() {
```

```
    console.log(`LoginPanelComponent : this.username :
      ${this.username}`);
    this.userService.authenticateUser(this.username,
      this.password).subscribe((response: Object) =>
    {
      console.log(`LoginPanelCompenent : response :
        ${JSON.stringify(response)}`);
      localStorage.setItem('token', response);
      this.router.navigate([``]);
    }, (err) => {
      console.log(`onLoginClicked() : error :
        ${JSON.stringify(err, null, 4)}`);
      this.error = `${err.message}`;
    }, () => {
      // наконец
      console.log(`finally.`);
    });
  }
```

Здесь мы обновили нашу функцию-конструктор, чтобы запросить экземпляр класса `UserService` наряду с классом `Router`. Затем мы обновили функцию `onLoginClicked` для вызова функции `authenticateUser` для `userService` и подписки (`subscribe`) на результаты Observable. Мы также определили все три функции успеха, ошибки и завершения для Observable. У функции успеха есть единственный параметр, который будет включать `response`, полученный от сервера Express. Мы записываем этот объект response в консоль, а затем создаем в `localStorage` элемент с именем `'token'` для хранения этого объекта. Потом мы используем функцию `navigate` для нашего объекта маршрутизатора, чтобы перенаправить браузер на главную страницу.

Наш обработчик ошибок запишет сообщение об ошибке в консоль, а также обновит сообщение об ошибке, отображаемое пользователю. Функция завершения просто записывает сообщение в консоль.

Обратите внимание, что мы используем локальное хранилище в качестве механизма для хранения состояния аутентификации. Мы могли бы использовать для этой цели куки, но HTML5 позволяет использовать локальное хранилище, что дает много преимуществ по сравнению с простыми куки. Например, куки отправляются и принимаются при каждом запросе, а это означает, что может быть затронут HTTP-трафик. Кроме того, локальное хранилище позволяет хранить локально 5 МБ информации вместо минимальных 4 КБ, которые позволяют куки.

Теперь, когда мы знаем, правильно ли пользователь вошел, мы можем обновить класс `AuthGuard`, чтобы проверить, есть ли у пользователя элемент в `localStorage` с именем `'token'`:

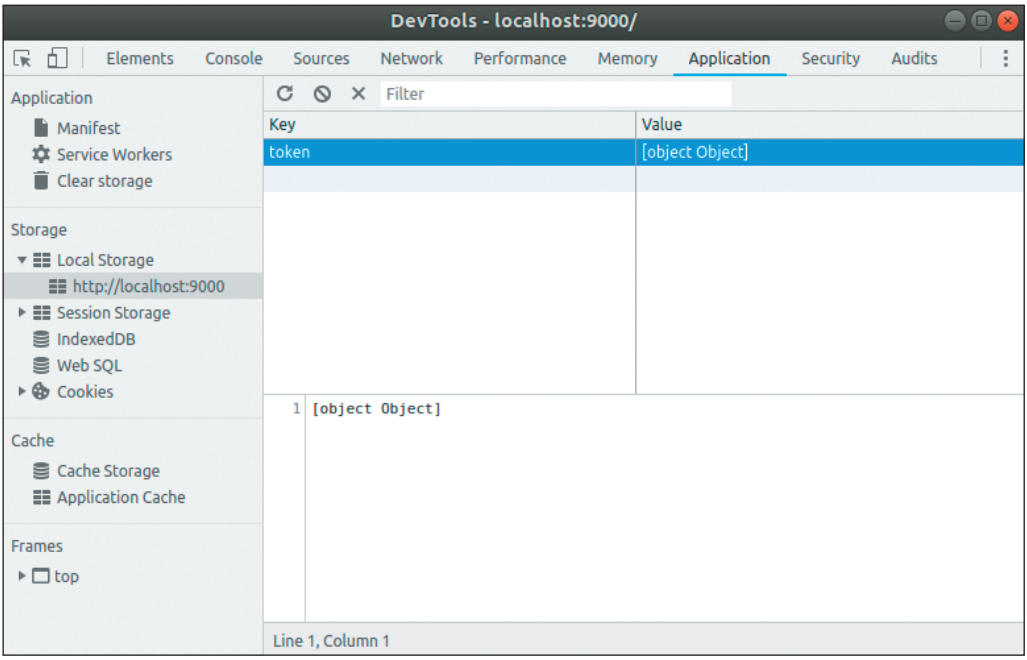
```
export class AuthGuard implements CanActivate {

constructor(private router: Router) { }

canActivate(
   route: ActivatedRouteSnapshot,
   state: RouterStateSnapshot): boolean
   {
      let token = <Object>localStorage.getItem('token');
      console.log(`Authguard : token : ${token}`);
      if (token) {
         return true;
      }
      return false;
   }
}
```

Здесь мы обновили функцию `canActivate` класса `AuthGuard`, чтобы проверить наличие элемента `'token'` в `localStorage`. Если он существует, мы записываем содержимое токена в консоль, а затем разрешаем доступ к этому маршруту, возвращая значение `true`. Если в `localStorage` ничего нет, мы возвращаем значение `false`.

После внесения этих изменений мы можем войти в приложение. Поток событий для этой последовательности выглядит следующим образом.

1. Ввести имя пользователя и пароль на экране входа.
2. Нажать на кнопку **Login**.
3. Обработчик `onLoginClicked` будет использовать `UserService` для отправки в обработчик входа Express.
4. Обработчик Express возвращает успешный ответ.
5. `LoginPanelComponent` сохраняет этот ответ в локальном хранилище.
6. Затем `LoginPanelComponent` перенаправляет браузер на страницу SecureComponent.
7. `AuthGuard`, который защищает доступ к `SecureComponent`, проверяет, может ли он найти элемент в локальном хранилище.
8. Таким образом, `AuthGuard` разрешает доступ к `SecureComponent`.

Обратите внимание, что в настоящее время у нас нет механизма выхода, после того как мы вошли. Локальное хранилище также остается после завершения работы браузера. Это означает, что даже перезапуск браузера не позволит нам выйти из приложения. В настоящее время мы можем нажать клавишу *F12*, чтобы перейти к удобным инструментам разработчика, и вручную удалить токен из локального хранилища, как показано на приведенном ниже скриншоте:



Использование JWT-токенов

Механизм безопасной аутентификации на предмет того, поступил ли запрос из авторизованного источника, был значительно упрощен благодаря введению стандарта **JSON Web Token** (**JWT**), который описан в открытом стандарте RFC 7519. Идея JWT заключается в том, что стандартный JSON-объект шифруется с использованием *секретного* ключа. Этот процесс шифрования известен как *подписание* токена. Только владелец *секретного* ключа может подтвердить, что этот токен действителен. Другими словами, сервер создает токен, используя свой секретный ключ, и тогда любой запрос, поступающий на сервер, может проверить, что токен был подписан правильно. Таким образом, токены JWT не имеют состояния и могут довольно легко использоваться в среде с балансировкой нагрузки. Пока все серверы в кластере с балансировкой нагрузки знают секретный ключ, все они могут проверять входящий токен, независимо от того, какой сервер его сгенерировал. Токены JWT могут быть настроены так, чтобы иметь время действия, и, следовательно, могут быть действительными в течение короткого периода времени.

В мире конечных точек REST без сохранения состояния мы должны обеспечить доступ к определенным конечным точкам только действительным пользователям наших систем. Мы можем легко настроить наши точки так, чтобы они требовали токен JWT, а затем проверить, что он был выдан с использованием правильного ключа, прежде чем принимать действие.

В этом разделе мы покажем, как выпускать и проверять токены JWT. Одна из популярных библиотек, доступных для работы с JWT-токенами, называется `jsonwebtoken`. Мы можем установить эту библиотеку для работы с JWT из npm:

```
npm install jsonwebtoken
```

Первая часть работы с JWT токеном – это его создание. Помните, что для создания токена нам нужен ключ шифрования (secret). Этот ключ не должен быть виден никому, кроме сервера, который создает токены, поэтому лучше всего, чтобы мы создавали токены на сервере Express.

Давайте обновим наш маршрутизатор входа Express, чтобы создать и подписать токен JWT:

```
import * as jwt from "jsonwebtoken";
const jwtSecret = '0e4253ef-5e4f-4d62-8eeb-c80e36a68c8a';

router.post(`/login`, (req: any, res: any, next: any) => {
   serverLog(`POST /login`);
   if (req.body.username && req.body.username.length > 0
      && req.body.password && req.body.password.length > 0) {
      let user_context = {
         username: req.body.username,
         token: ''
      }

      var token = jwt.sign(user_context, jwtSecret);
      user_context.token = token;
      res.json(user_context);
   } else {
      serverLog(`/login - Error : Invalid username or password`);
      res.status(401).send('Invalid username or password');
   }
});
```

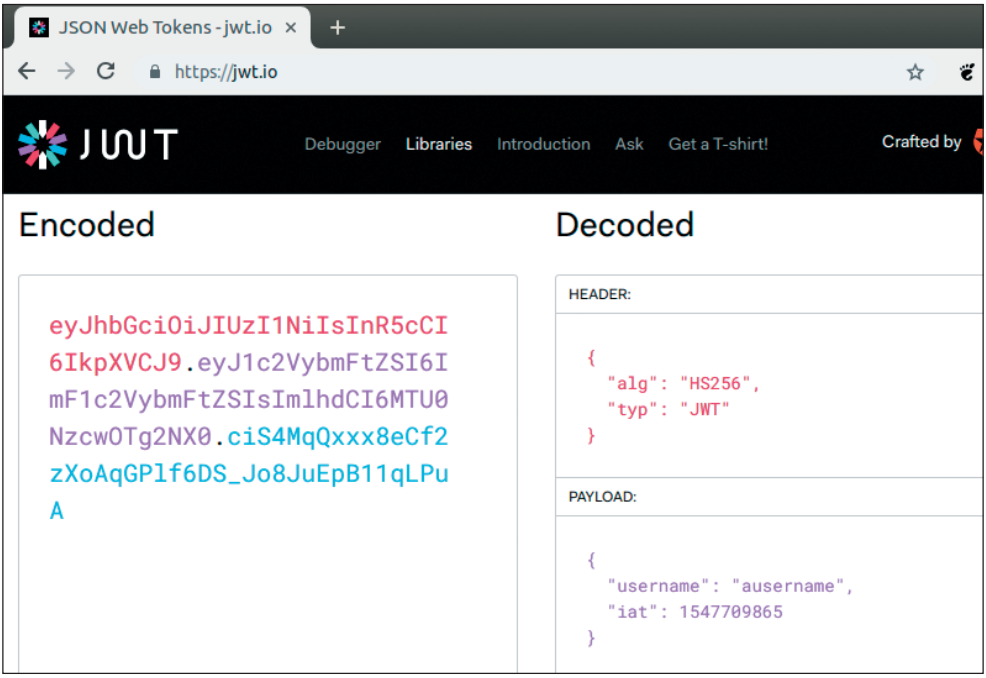
Здесь мы импортировали библиотеку `jsonwebtoken` и назвали ее `jwt`. Затем мы создаем переменную `jwtSecret` и присваиваем ей строку `GUID`. Это ключ шифрования, который сервер будет использовать для подписи JWT-токена.

В нашем обработчике post мы создаем объект JavaScript, у которого есть свойство username. Это будет содержимым нашего токена. Затем мы используем функцию `sign` из библиотеки `jwt`, передавая объект JavaScript в качестве первого аргумента и post в качестве второго объекта. Как только объект подписан, мы просто возвращаем его. Это все, что нужно для создания токена JWT. Мы просто создаем стандартный объект JavaScript, а затем используем библиотеку для шифрования и подписи объекта.

Давайте войдем в наше приложение и посмотрим на этот токен. Если мы перейдем к нашим инструментам разработчика и откроем вкладку **Application** (**Приложение**), то увидим токен, который хранится в локальном хранилище. Это просто строка:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6ImF1c2Vy
bmFtZSIsImlhdCI6MTU0NzcwOTg2NX0.ciS4MqQxxx8eCf2zXoAqGPlf6DS_
Jo8JuEpB11qLPuA
```

Интересная вещь касательно этих токенов заключается в том, что любой может их декодировать, но только сервер с ключом шифрования, который использовался для подписи, может верифицировать токен. В качестве примера того, как декодировать этот токен, давайте перейдем на сайт `jwt.io` и просто вставим эту строку в текстовое поле Encoded:



Здесь видно, что JWT-токен был декодирован, а с правой стороны показана фактическая полезная нагрузка. Эта полезная нагрузка включает в себя наше свойство username, а также свойство iat, которое указывает, когда истечет срок действия токена. Обратите внимание, что нам не нужно было давать сайту знать ключ шифрования, чтобы он расшифровал токен. Это означает, что любой может декодировать токен, но только сервер с секретным ключом может это проверить.

Открытая природа токенов JWT означает, что мы не можем хранить конфиденциальную информацию в токене. Например, если бы нам нужно было сохранить

свой пароль, любой бы мог получить доступ к токену и расшифровать его, чтобы узнать пароль.

Верификация токенов

Опять же, когда токен используется, сервер, который сгенерировал токен, обязан убедиться, что он действителен, перед тем как его использовать. Давайте создадим обработчик POST на нашем сервере Express, который будет проверять токен:

```
router.post(`/validate-user`, (req: any, res: any, next: any) => {
   serverLog(`POST /validate-user`);
   console.log(`req.body : ${JSON.stringify(req.body)}`);

   if (req.body.token && req.body.token.length > 0) {
      try {
         let verifiedJwt = jwt.verify(req.body.token, jwtSecret);
         return res.json(verifiedJwt);
      } catch (err) {
         serverLog(`/validate-user : token error`);
         res.status(401).send('invalid auth token');
      }
   } else {
      serverLog(`/validate-user : token not found error`);
      res.status(401).send('Invalid auth token');
   }
});
```

Здесь мы определили обработчик маршрута для POST к конечной точке с именем `validate-user`.

Этот обработчик проверяет наличие полезной нагрузки токена, а затем вызывает функцию `verify` в библиотеке `jwt`. Эта функция принимает сам токен в качестве входных данных, а также ключ шифрования, который мы использовали для подписи токена. Обратите внимание, что мы обернули этот вызов в блок `try catch`, поскольку библиотека `jwt` выбросит исключение, если токен не может быть верифицирован правильно. Если токен действителен, мы просто возвращаем его содержимое в виде структуры JSON. Если токен нельзя верифицировать, мы возвращаем код состояния **401 HTTP** с сообщением об ошибке.

Наши две конечные точки Express успешно создали или подписали токен, а также верифицировали его. В обоих случая был использован один и тот же секретный ключ.

Обратите внимание на важность этого шага проверки. Токен представляет собой простую строку и может быть довольно легко декодирован. Это означает, что его также очень легко изменить. Даже сайт `jwt.io` может изменить содержимое то-

кена и создать новую зашифрованную строку. Однако после того, как токен был изменен, он не пройдет проверку. Это означает, что невозможно восстановить модифицированный токен без секретного ключа. Поэтому каждый раз, когда серверу необходимо что-то делать от имени пользователя, он должен проверять, что предоставленный токен не был каким-то образом подделан.

Использование Observables в гневе – of, pipe и map

Теперь у нас есть конечная точка REST, которая может проверить наш токен. Далее нам нужно вызвать эту конечную точку из нашего приложения Angular, чтобы убедиться, что у пользователя, выполнившего вход, есть действительный токен. Помните, что при входе мы сохраняем этот токен в локальном хранилище и перенаправляем его в наш `SecureComponent`. `SecureComponent` защищен `AuthGuard`, который проверяет наличие этого токена. Поэтому нам нужно связаться с нашим сервером Express, сгенерировавшим этот токен, чтобы его проверить.

Наиболее очевидное место для размещения этого кода – сам `AuthGuard`, поэтому любая попытка получить доступ к `SecureComponent` должна будет проверить токен путем отправки с помощью метода `POST` на наш сервер Express, прежде чем продолжить. Опять же, мы не должны обращаться к конечным точкам REST из нашего кода напрямую, поэтому нам понадобится служба, которая может связаться с сервером от нашего имени. Поскольку для этой цели у нас уже есть `UserService`, мы можем просто создать в этой службе еще одну функцию для фактического вызова сервера Express:

```
validateUser(token: Object): Observable<Object> {
   const headers = new HttpHeaders();
   headers.append('Content-Type', 'application/json');
   const payload = {
      token: token
   };
   return this.httpClient.post(
      '/validate-user', payload,
      { 'headers': headers });
}
```

Здесь мы создали функцию `validateUser`, которая принимает единственный параметр с именем `token` типа `Object` и возвращает `Observable` типа `Object`. В рамках этой функции мы просто создаем POST-запрос к конечной точке `validate-user` с токеном в качестве содержимого. Это достаточно просто и имитирует то, что мы делали для конечной точки `login`.

Наши изменения в AuthGuard – вот где все становится немного сложнее. Мы обновим функцию canActivate:

```
canActivate(route: ActivatedRouteSnapshot,
state: RouterStateSnapshot):
   Observable<boolean>
{
   let token = <Object>localStorage.getItem('token');
   console.log(`Authguard : token : ${token}`);
   return this.userService.validateUser(token).pipe(
      map( (e: Object) : boolean => {
         console.log(`Authguard : e ; ${JSON.stringify(e)}`)
         return true;
      } ),

      catchError((err) => {
         // Ошибки 401 будут перехватываться здесь
         this.router.navigate(['/login']);
         return of(false);
      })
   );
}
```

Здесь есть несколько модификаций функции `canActivate`. Во-первых, мы не просто возвращаем логическое значение, а возвращаем `Observable` логического значения.

Во-вторых, мы обновили эту функцию, чтобы вызвать функцию `validateUser` из `UserService`, дабы отправить POST в конечную точку REST и проверить наш токен. Третья модификация, которую следует отметить, заключается в том, что мы используем метод `pipe` вместо метода `subscribe`. Мы подробно обсудим этот метод чуть позже. В методе `pipe` у нас есть вызов метода map, а также метода `catchError`. Опять же, мы подробно обсудим эти методы далее.

С точки зрения логического потока, то, что мы делаем в этой функции, выглядит следующим образом:

1) получение токена из локального хранилища;
2) вызов функции validateUser для UserService и передача токена;
3) это выдаст POST нашей конечной точке и вернет либо правильный ответ, либо ошибку;
4) если возвращается правильный ответ, возвращается значение true, чтобы AuthGuard был успешен;
5) если возникает ошибка, переходим по URL-адресу /login и возвращаем значение `false`, чтобы AuthGuard завершился ошибкой.

Синтаксис этой функции выглядит довольно жутковато и может привести к путанице, поэтому давайте разберем его на более мелкие части и обсудим каждый фрагмент по отдельности.

Функция `canActivate` сейчас возвращает тип `Observable<boolean>` вместо простого старого логического значения. Давайте посмотрим, как преобразовать логическое значение в Observable логического значения.

Чтобы обернуть логическое значение или любое другое значение в Observable, можно использовать функцию `of`. В качестве примера рассмотрим следующий код:

```
function usingObservableOf(value: number): Observable<boolean> {
   if (value > 10) {
      return of(true);
   } else {
      return of(false);
   }
}
```

Здесь мы определили функцию `usingObservableOf`, которая принимает число в качестве единственного параметра. Он возвращает тип `Observable <boolean>`, что означает, что он должен возвращать Observable. Тело функции проверяет, является ли переданный аргумент `value` больше `10`. Если это так, мы возвращаем `of(true)`, что, по сути, превращает наше логическое значение `true` в `Observable <boolean>`. Использование функции `of` можно увидеть в функции `canActivate`, в секции `catchError`.

Тело функции `canActivate` извлекает токен, хранящийся в локальном хранилище, а затем вызывает функцию `validateUser` в `UserService`. Однако у нас есть проблема, которая состоит в том, что функция `validateUser` возвращает тип `Observable<Object>`, который будет содержать JSON, возвращаемый конечной точкой `/validate-user`. К сожалению, мы не можем просто вернуть этот результат из функции `canActivate`, так как нам нужно вернуть тип `Observable <boolean>`. Чтобы разрешить эту ситуацию, мы используем функцию `pipe` и операторы `map` и `catchError` из RxJS.

Функция `pipe` используется для объединения двух функций в одну и выполняет каждую из этих функций последовательно, возвращая одну функцию. Таким образом, вместо использования `subscribe` для функции `validateUser` мы создаем новую функцию, используя `pipe`, которая объединяет функции `map()` и `catchError()`. Вот урезанная версия того, как это выглядит вместе:

```
function usingPipe( subject: Observable<Object>) {
   subject.pipe(
      map(() => {}),
      catchError((err) => {
         return of(false);
      })
   );
}
```

Здесь у нас есть функция `usingPipe`, которая принимает единственный параметр с именем `subject` типа `Observable<Object>`. Затем мы используем функцию `pipe` для аргумента `subject` и связываем две функции вместе.

Первая функция создается из функции `map`, а вторая – из функции `catchError`. Обратите внимание, что функции `map` и `catchError` используют функцию в качестве исходного аргумента. Там, где функция `map` будет автоматически возвращать `Observable`, функция `catchError` этого не делает. Вот почему нам нужно использовать функцию `of` для создания Observable из логического значения `false`.

Функция map или, вернее, оператор map из библиотеки RxJS используется для преобразования каждого элемента в Observable с использованием функции. В качестве простого примера рассмотрим следующий код:

```
function usingMap(subject: Observable<Object>):
   Observable<boolean> {
   return subject.pipe(
      map((object: Object): boolean => {
         return false;
   }));
}
```

Здесь у нас есть функция `usingMap` с единственным параметром `subject` типа `Observable<Object>`. Однако эта функция возвращает тип `Observable <boolean>`. Это означает, что каждый из элементов входного потока Observable нужно преобразовать из `Object` в `boolean`. Мы используем функцию `pipe` непосредственно для аргумента `subject` и предоставляем простую функцию для использования `map`. Обратите внимание на сигнатуру этой анонимной функции. Она принимает исходный тип `Object`, а возвращает логическое значение. Вот так мы можем использовать `pipe` и `map` для преобразования потока Observable из одного типа в другой.

Функция `canActivate` объединяет функции `pipe`, `map` и `catchError` в единый рабочий поток для преобразования результата вызова REST validateUser из `Observable<Object>` в `Observable <boolean>`.

Поэтому наша новая версия `AuthGuard` вызывает конечную точку REST в `/validate-user` и отправляет токен, полученный из локального хранилища. Если конечная точка возвращает успех, то `AuthGuard` знает, что использованный токен был успешно проверен, и разрешит доступ к `SecureComponent`.

Итак, мы увидели, как создать JWT-токен, как его декодировать и как верифицировать. Мы также увидели, как можно использовать возможности Observables для преобразования данных из одного типа в другой.

Внешняя аутентификация

Как мы упоминали в начале главы, большинство современных сайтов позволяет регистрироваться через сервисы внешней аутентификации, такие как Facebook, Google или LinkedIn. Это означает, что эти сервисы проверяют учетные данные пользователя, а затем пересылают нашему приложению подробности о пользователе. Эти данные обычно включают в себя имя пользователя, адрес электронной почты и, возможно, даже ссылку на изображение, которое мы можем использовать на нашем сайте. Это довольно простая замена стандартного процесса регистрации, где мы собираем и храним информацию о пользователе и его пароле. Использование сервисов внешней аутентификации также делает наши сайты более привлекательными и доступными для более широкого круга пользователей.

В этом разделе мы будем интегрироваться с Google, чтобы каждый, у кого есть действующая учетная запись в Google, имел доступ к нашему сайту. Библиотеки и процессы, используемые для внешней аутентификации, очень похожи у основных поставщиков услуг аутентификации, поэтому мы сосредоточимся на аутентификации Google.

Получение API-ключа Google

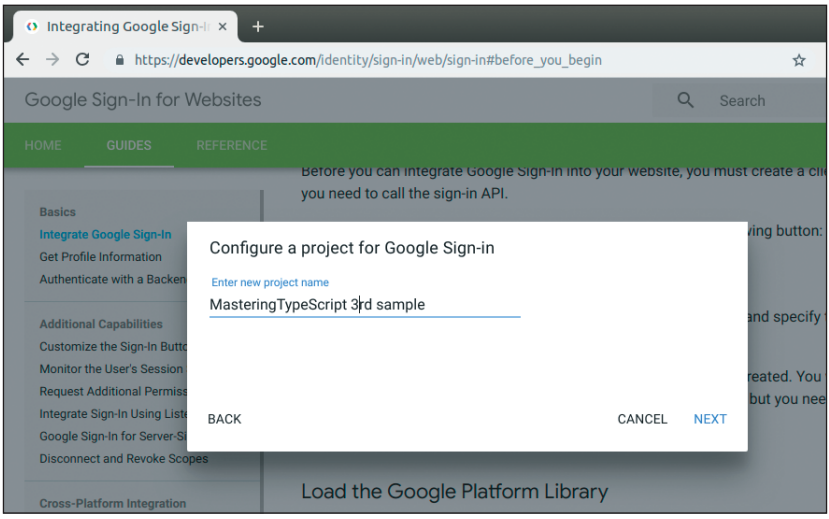
Чтобы облегчить интеграцию провайдеров аутентификации, мы можем обратиться к открытым исходным кодам и использовать библиотеку Angular под названием `angular-6-social-login-v2`. Ее можно легко установить через `npm`, как обычно:

```
npm install angular-6-social-login-v2
```

После установки нам нужно будет зарегистрировать наше приложение в Google и сгенерировать API-ключ Google. Документация по пакету social login в Angular содержит ссылку на следующую страницу, которую мы можем использовать для генерации ключа: `https://developers.google.com/identity/sign-in/web/sign-in#before_ you_ begin`.

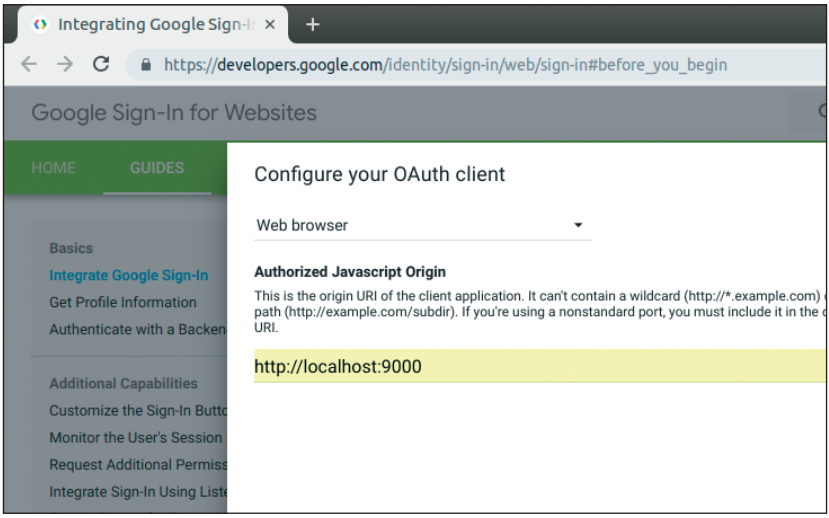
Чтобы сгенерировать ключ, нам потребуется действующая учетная запись в Google, а затем просто следуйте инструкциям на этой странице, которые проведут нас через процесс генерации ключа.

Нажмите кнопку **Configure a Project** (**Настроить проект**) и введите имя своего проекта:



Здесь мы ввели имя **MasteringTypeScript 3rd sample**. Это имя может быть любым. После ввода нажмите кнопку **NEXT** (**Далее**), чтобы перейти к следующему шагу. На следующем шаге будет запрошено имя для настройки вашего клиента Oauth, которое появится в окне согласия пользователя. После этого последняя часть информации, которая требуется, – это настроить клиента Oauth.

Выберите веб-браузер в качестве источника из раскрывающегося списка, а затем введите URL-адрес, который будет использоваться веб-сервером. В целях разработки мы можем ввести здесь `http://localhost: 9000`, который является базовым URL-адресом нашего приложения, как показано на приведенном ниже скриншоте:



После настройки нам будет предоставлен идентификатор клиента и ключ для шифрования. Скопируйте эти значения в безопасное место, так как мы будем использовать их позже.

Настройка социального логина

Процесс интеграции библиотеки социального логина в Angular не сложен и разбит на две части. Во-первых, нам нужно будет настроить поставщика, а во-вторых, нам нужно будет использовать этого поставщика для получения токена пользователя. Различные социальные провайдеры настроены как службы и как таковые требуют изменений в файле `app.module.ts`:

```
import {
   SocialLoginModule,
   AuthServiceConfig,
   GoogleLoginProvider
} from 'angular-6-social-login-v2';

export function getAuthServiceConfig(): AuthServiceConfig {
   let config = new AuthServiceConfig([{
      id: GoogleLoginProvider.PROVIDER_ID,
      provider: new GoogleLoginProvider(`<your client id goes here>`)
   }]);
   return config;
}

@NgModule({
   declarations: [
      ...существующие объявления...
   ],
   imports: [
      SocialLoginModule,
      ...существующие модули...
   ],
   providers: [
      {
         provide: AuthServiceConfig,
         useFactory: getAuthServiceConfig
      }
   ],
   bootstrap: [AppComponent]
})
export class AppModule { }
```

Здесь мы импортируем классы `SocialLoginModule`, `AuthServiceConfig` и `GoogleLoginProvider` из модуля `angular-6-social-login-v2`. Затем мы создаем функцию `getAuthServiceConfig`, которая возвращает экземпляр но-

вого класса `AuthServiceConfig` с массивом конфигураций поставщика услуг. Каждому элементу массива требуются свойства `id` и `provider`. Свойство `id` просто использует одно из значений поставщиков, которые включены в библиотеку, в данном случае это `GoogleLoginProvider.PROVIDER_ID`. Для Facebook мы будем использовать `FacebookLoginProvider.PROVIDER_ID`. Свойство `provider` создает экземпляр поставщика, инициализируя его, используя идентификатор клиента, который мы получили при регистрации нашего приложения в Google. Обязательно замените строку `<your client id goes here>` идентификатором клиента, который мы получили от Google.

Затем нам нужно обновить свойство нашего модуля import, чтобы включить в него `SocialLoginModule`, а также создать запись свойства `providers` для обращения к нашей фабричной функции `getAuthServiceConfig`. У записи `providers` есть два свойства: `provide` и `useFactory`. Эти свойства используются для настройки `AuthService`, которую мы будем использовать для подключения к службе аутентификации Google.

На этом настройка наших внешних поставщиков аутентификации окончена. Мы просто настроили фабричный метод для правильной установки идентификатора клиента, который Google должен будет подтвердить, что мы зарегистрировались для доступа к API.

Использование данных пользователя Google

Теперь мы можем сосредоточиться на включении логина Google в `LoginPanelComponent`. Для начала нам нужно установить обработчик кликов для кнопки **Login with Google**:

```
<button class="btn btn-primary"
  (click)="onLoginGoogleClicked()">Login with Google</button>
```

Здесь мы обновили файл `login-panel.component.html` и связали функцию `onLoginGoogleClicked` с событием `click` в нашей форме. Затем мы можем обновить сам компонент:

```
constructor(
  private userService: UserService,
  private router: Router,
  private socialAuthService: AuthService) { }
  onLoginGoogleClicked() {
    this.socialAuthService.signIn(GoogleLoginProvider.PROVIDER_ID)
      .then((userdata) => {
        this.userService.authenticateGoogleUser(userdata)
        .subscribe((response: any) => {
```

```
            localStorage.setItem('token', response);
            this.router.navigate([``]);
          }, (err) => {
            console.log(`onLoginClicked() : error :
              ${JSON.stringify(err, null, 4)}`);
            this.error = `${err.message}`;
          });
        }).catch((error) => {
          console.log(`error : ${error}`);
    });
  }
```

Здесь мы включили закрытую переменную конструктора `socialAuthService`, которая будет установлена в экземпляр `AuthService` с помощью фреймворка внедрения зависимостей Angular. Затем мы определяем функцию `onLoginGoogleClicked`, которая использует этот `AuthService` для вызова функции `signIn` с правильным `PROVIDER_ID` для Google. Обратите внимание, что этот вызов возвращает промис, поэтому мы определили обработчик `.then` для успешного ответа и обработчик `.catch` в случае ошибки.

Когда служба аутентификации Google вернется с успешным ответом, она предоставит нам структуру в формате JSON, которая включает в себя информацию, связанную с вошедшим в систему пользователем. Затем мы передаем эти пользовательские данные в новую функцию нашего `UserService`, которая отправит POST в новую конечную точку, чтобы сгенерировать локально подписанный JWT-токен. Получив подписанный токен, мы можем сохранить его в локальном хранилище и перенаправить пользователя в `SecureComponent`.

Реализация `authenticateGoogleUser` – это простой метод POST `HttpClient` в `UserService` для отправки пользовательских данных на наш сервер Express:

```
authenticateGoogleUser(data: any) {
  const headers = new HttpHeaders();
  headers.append('Content-Type', 'application/json');

  return this.httpClient.post(
    '/login-google', data,
    { 'headers': headers });
}
```

Эта функция просто создает POST для конечной точки Express с именем `/login-google` и отправляет пользовательские данные в виде тела JSON. Мы можем реализовать обработчик маршрута в файле `userRoutes.ts` сервера Express следующим образом:

```
router.post(`/login-google`, (req: any, res: any, next: any) => {
  serverLog(`POST /login-google`);
```

```
    if (req.body.name & req.body.name.length > 0) {
       let user_context = {
          username: req.body.name
       }

       var token = jwt.sign(user_context, jwtSecret);
       res.json(token);
    } else {
       serverLog(`/login-google - Error : Invalid google token`);
       res.status(401).send('Invalid google token');}
});
```

Здесь мы определили обработчик POST для конечной точки с именем `/login-google`. Этот обработчик просто проверяет, является ли допустимым свойство name опубликованных данных в формате JSON. Если это так, он создает объект `user_context`, который содержит одно свойство с именем `username`, и устанавливает его значение равным значению свойства name. Как и в случае с обработчиком маршрута `/login`, мы создаем JWT-токен и подписываем его с помощью функции `sign` библиотеки `jwt`, передавая ключ шифрования нашего сервера. Затем мы возвращаем этот токен вызывающей стороне конечной точки REST.

Итак, чего мы достигли в отношении внешней службы аутентификации, такой как Google? Вот шаги, которые мы проработали.

1. Установить библиотеку `angular-6-social-login-v2`.
2. Зарегистрировать приложение в Google и получить идентификатор клиента и ключ для шифрования.
3. Обновить файл `app.module.ts` для настройки и регистрации поставщика `AuthServiceConfig`.
4. Создать событие обработчика кликов, когда пользователь нажимает кнопку **Login with Google**.
5. Вызвать функцию `signIn` экземпляра `AuthService` для аутентификации пользователя с помощью Google и вернуть структуру пользовательских данных в формате JSON.
6. Отправить эту структуру на сервер Express с помощью службы Angular.
7. Сгенерировать токен JWT, подписанный сервером Express.
8. Сохранить этот токен в локальном хранилище, чтобы `AuthGuard` мог позволить пользователю перейти к `SecureComponent`.

Резюме

В этой главе мы рассмотрели некоторые фундаментальные строительные блоки, участвующие в разработке приложений. Мы начали с создания сервера Express, способного обслуживать наше приложение Angular, а затем изучили, как мы можем устанавливать и использовать переменные конфигурации и ведение журна-

ла сервера для создания готового к эксплуатации сервера Express. Затем мы исследовали использование Brackets и Emmet, чтобы облегчить быстрое проектирование макетных веб-страниц.

Однако основная часть этой главы была посвящена аутентификации. Мы рассмотрели использование маршрутизации в Angular и Auth Guards, чтобы определить безопасный раздел приложения, который доступен только зарегистрированным пользователям системы. Затем мы работали над тем, как отправить запросы `POST` и `GET` в конечную точку REST с помощью `HttpClient` и маршрутизации Express.

Для правильной защиты нашего приложения мы реализовали генерацию и верификацию токенов JWT. Наконец, мы исследовали использование поставщика внешней аутентификации, чтобы пользователи нашего сайта могли войти в систему с помощью Google.

В следующей главе мы объединим все наши приобретенные знания и создадим полноценное приложение в Angular, которое включает в себя все строительные блоки, с которыми мы работали на протяжении всей книги. Мы даже найдем время, чтобы познакомиться с новыми шаблонами проектирования, которые помогут нам при создании полноценного приложения. Мы будем работать с данными в формате JSON, писать модульные и приемочные тесты, чтобы охватить как можно больше функциональности и исследовать дополнительные возможности Observables.

Глава 14

Переходим к практике

В заключительной главе мы будем использовать методы и принципы, которые изучили до настоящего момента, для создания веб-приложения. Это приложение будет использовать дизайн панели слева направо, который мы рассмотрели в `главе 11` *«Объектно-ориентированное программирование»*, где используются шаблоны State и Mediator для контроля состояния экрана. Мы продолжим взаимодействие с конечными точками REST, как мы обсуждали в `главе 13` *«Создание приложений»*. Мы реализуем полностью работающий API REST с использованием Express, который будет считывать данные из базы данных backend. Затем мы рассмотрим новые методы работы с Observables, которые позволят нам объединять данные, фильтровать их и координировать асинхронные запросы к API REST. Мы также обсудим и реализуем шаблон проектирования Domain Events, чтобы помочь взаимодействию между независимыми компонентами приложения.

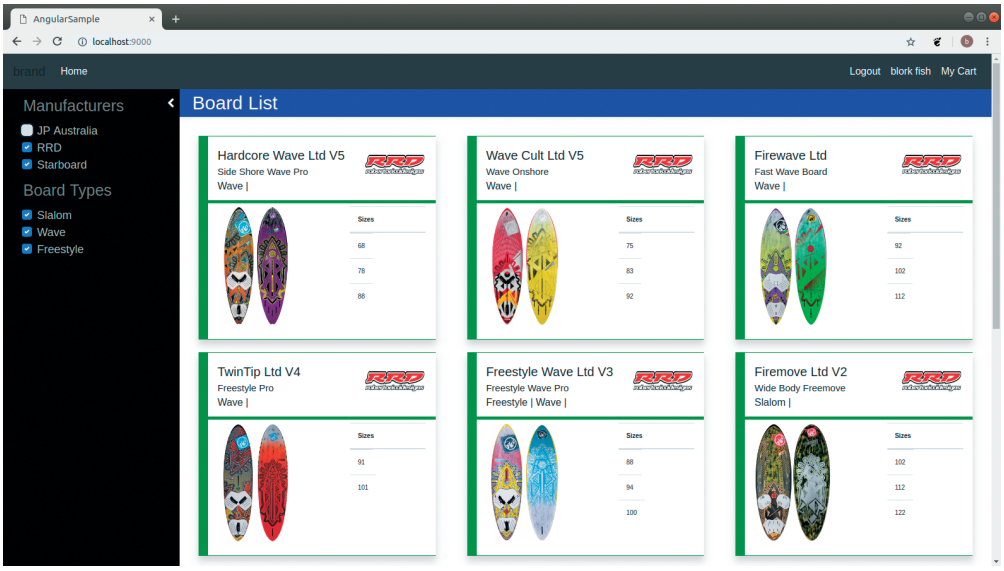
У нас уже есть в руках все строительные блоки и глубокие познания, необходимые для того, чтобы собрать приложение, поэтому эта глава посвящена повторному использованию и интеграции компонентов. Для этой главы мы расширим образец приложения на Angular, над которым мы работали и который в последний раз обновляли в предыдущей главе *«Создание приложений»*.

В этой главе будут рассмотрены следующие темы:

- обзор приложения;
- структура базы данных;
- построение управляемых базой данных конечных точек с помощью Express;
- принципы проектирования конечных точек API REST;
- работа с конечными точками REST API в Angular;
- использование функции `concatMap`;
- применение функции `forkJoin`;
- шаблон проектирования Domain Events;
- создание и использование Domain Events;
- использование функции `filter`.

Приложение Board Sales

Мы создадим простое приложение под названием **Board Sales**, в котором будет представлен список досок для виндсерфинга на главной странице, а затем пользователь сможет просмотреть подробную информацию о любой из предлагаемых досок, нажав на нее. При нажатии на конкретную доску панель с подробной информацией о доске сдвинется влево. Мы также будем использовать панель навигации слева, чтобы предоставить пользователю опции для фильтрации списка досок. Если пользователь нажимает на определенный фильтр, то диапазон отображаемых досок будет отфильтрован в соответствии с этим выбором. Главная страница будет выглядеть так:

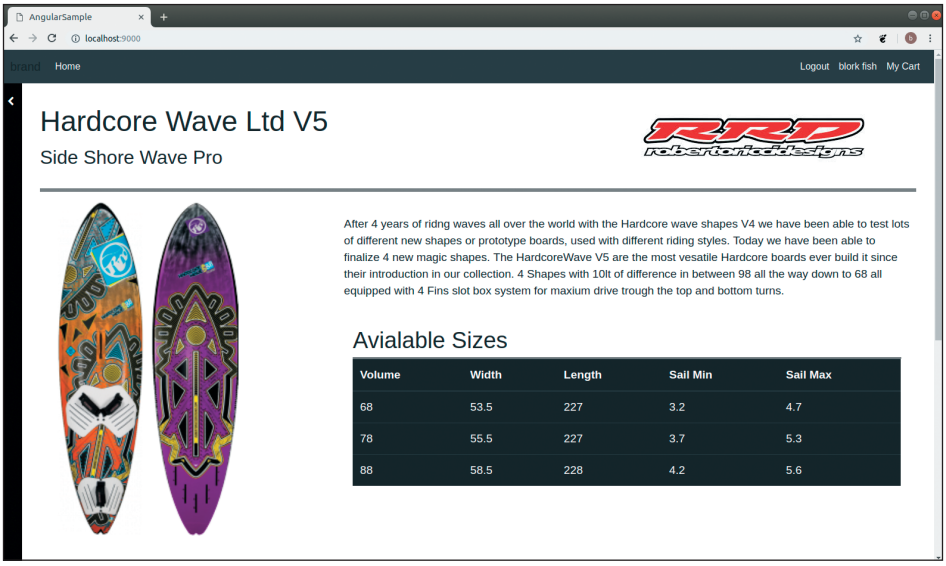


Здесь видно, что у нас есть верхняя панель навигации, панель навигации с левой стороны, показывающая параметры фильтра, и главная панель, на которой показаны доски.

Современные доски для виндсерфинга бывают разных размеров и измеряются по объему. Как видно из списка, рядом с каждой доской указана таблица доступных размеров. Меньшие по объему доски обычно используются для парусного спорта, а доски большего объема используются для гонок или слалома. Доски, которые находятся между ними, можно классифицировать как доски для фристайла. Они используются для выполнения акробатических трюков на водном зеркале. У каждой доски есть производитель, который соответствует логотипу, указанному рядом с каждой доской. Наша панель фильтров слева позволяет пользователю выбрать либо производителя, либо тип доски. Таким образом, чтобы просмотреть только доски, изготовленные **RRD**, пользователь может нажать на фильтр **RRD**

под словом **Manufacturers**. Аналогично, чтобы отфильтровать список досок по доскам для слалома, пользователь может щелкнуть фильтр **Slalom** на панели слева.

При нажатии на любую конкретную доску откроется экран с подробной информацией о доске:



Здесь у нас есть подробное представление доски с увеличенным изображением и более подробное описание самой доски, а также расширенная таблица доступных размеров.

Одним из важных аспектов выбора доски для виндсерфинга является диапазон размеров парусов, которые она поддерживает. При очень сильном ветре паруса меньшего размера используются для того, чтобы позволить виндсерферу контролировать силу ветра. Аналогичным образом, при более слабом ветре, большие паруса используются для генерации большей силы. Комбинация размера доски, ее типа и доступных размеров парусов – все это используется для выбора правильной доски для спортсмена и условий плавания. В подробном представлении перечислены минимальные и максимальные размеры парусов, которые может поддерживать доска.

API на основе базы данных

Чтобы предоставить данные, необходимые для этого приложения, мы будем хранить и извлекать данные из традиционной базы данных. Мы предоставим конечную точку REST для каждой таблицы нашей базы данных, которая нам понадо-

бится. Node поддерживает широкий спектр баз данных, в том числе традиционные реляционные базы данных, такие как Oracle или SQL Server, или объектные базы данных, такие как MongoDB или CouchDB. В этом упражнении мы будем использовать Sqlite3 в качестве резервной базы данных.

Sqlite3 – это очень маленький, быстрый и автономный движок базы данных. Он поддерживает запросы языка SQL и структуры стандартных таблиц и содержится в одном файле на диске или даже в виде временной базы данных в памяти. Его можно установить на Windows, Linux и MacOS, и он занимает очень мало места. В Sqlite3 также есть несколько очень простых, но мощных инструментов с графическим интерфейсом, которые помогут в проектировании базы данных и вводе данных, таких как DB Browser для SQLite (`http://sqlitebrowser.org/`).

Структура базы данных

Основные таблицы, которые нам понадобятся для нашего приложения, включают в себя таблицы `Board`, `Manufacturer` и `BoardType`. Таблица Board выглядит так:

```
CREATE TABLE 'Board' (
   'id' INTEGER NOT NULL DEFAULT 1 PRIMARY KEY
     AUTOINCREMENT UNIQUE,
   'name' TEXT NOT NULL,
   'short_description' TEXT,
   'long_description' TEXT,
   'img' TEXT
);
```

Здесь мы видим команду `CREATE`. У нее есть свойство `id`, которое будет автоматически увеличиваться, и оно также служит первичным ключом. Наряду со свойством `id` у каждой доски есть свойства `name`, `short_description`, `long_description` и `img`. Свойство `img` будет использоваться для загрузки соответствующего изображения для отображения.

Таблица `Manufacturer` выглядит следующим образом:

```
CREATE TABLE 'Manufacturer' (
   'id' INTEGER NOT NULL DEFAULT 1 PRIMARY KEY
     AUTOINCREMENT UNIQUE,
   'name' TEXT NOT NULL,
   'logo' TEXT NOT NULL
);
```

Здесь видно, что каждая запись в таблице `Manufacturer` будет иметь свойства `id`, `name` и `logo`. Чтобы связать строку в таблице `Board` с таблицей `Manufacturer`, нам понадобится таблица отображения `BoardManufacturer`:

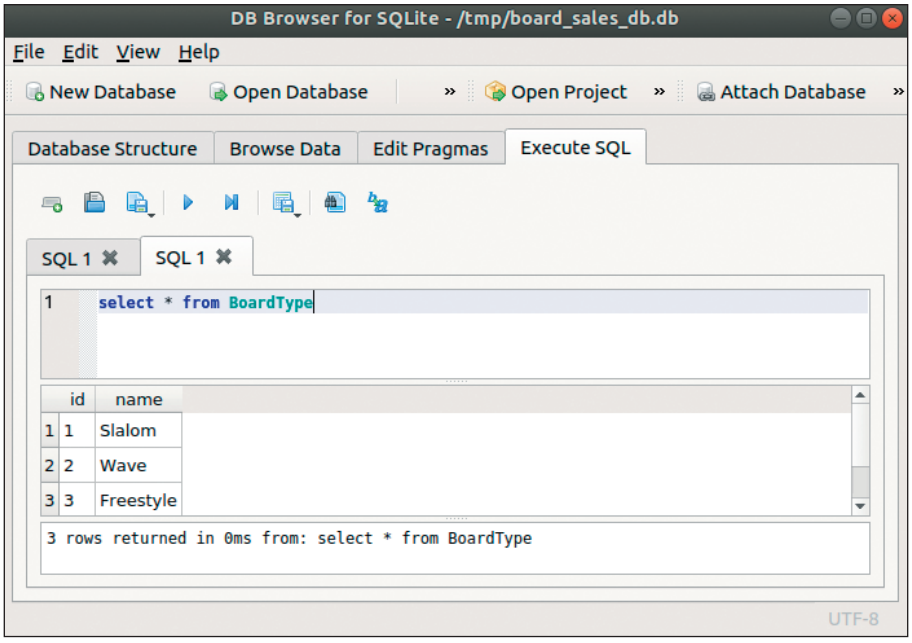
```
CREATE TABLE 'BoardManufacturer' (
   'id' INTEGER NOT NULL DEFAULT 1 PRIMARY KEY
     AUTOINCREMENT UNIQUE,
   'board_id' INTEGER NOT NULL,
   'manufacturer_id' INTEGER NOT NULL
);
```

Здесь видно, что каждая строка в таблице `BoardManufacturer` будет иметь свойства `board_id` и `factory_id`. Мы будем использовать эту таблицу для связи конкретной доски с ее производителем.

Таблица `BoardType` очень похожа на таблицу `Manufacturer`:

```
CREATE TABLE 'BoardType' (
   'id' INTEGER NOT NULL DEFAULT 1 PRIMARY KEY
     AUTOINCREMENT UNIQUE,
   'name' TEXT NOT NULL
);
```

Здесь мы будем хранить свойства `id` и `name` для типов досок, которые нам понадобятся, как показано на приведенном ниже скриншоте:



Видно, что таблица `BoardType` содержит три записи для типов досок: `Slalom`, `Wave` и `Freestyle`.

Чтобы связать доску с ее типом, мы снова создадим таблицу сопоставления:

```
CREATE TABLE 'BoardBoardType' (
   'id' INTEGER NOT NULL DEFAULT 1 PRIMARY KEY
     AUTOINCREMENT UNIQUE,
   'board_id' INTEGER NOT NULL,
   'board_type_id' INTEGER NOT NULL
);
```

Здесь мы создадим таблицу с именем `BoardBoardType`, которая будет содержать записи для каждой доски, которая есть в нашей базе данных, и список ее типов. Обратите внимание, что некоторые доски можно отнести к нескольким типам, поэтому это могут быть как доски для катания по волнам, так и доски для фристайла. В этом случае таблица `BoardBoardType` будет содержать две записи для одного и того же `board_id`.

Обратите внимание, что в традиционных базах данных внешние ключи и уникальные индексы используются для поддержания ссылочной целостности данных в базе данных. Для краткости мы не включили ни одну из этих мер безопасности, но в действительности они должны применяться при работе с любой производственной базой данных. Пожалуйста, обратитесь к примеру кода, который сопровождает эту главу, чтобы получить полную базу данных Sqlite3.

Конечные точки API

Чтобы предоставить данные, которые мы сохранили в нашей базе данных, нашему приложению, мы создадим набор конечных точек REST на нашем сервере Express, который подключится к нашей базе данных, выполнит SQL-запрос и вернет данные в виде объектов JSON. Для того чтобы работать с базой данных Sqlite3, мы будем использовать библиотеку `sqlite3`, которую можно установить так:

```
npm install sqlite3
npm install @types/sqlite3 -saveDev
```

Чтобы передать данные нашему приложению, мы создадим новый обработчик Express в каталоге `express/routs` с именем `dataRoutes.ts`:

```
import * as express from 'express';
import { serverLog } from '../main';

var router = express.Router();

router.get(`/boards`, async (req: any, res: any, next: any) => {
   serverLog(`GET /boards`);
   res.json({ result: 'success' })
});

export {router};
```

Здесь у нас есть стандартный обработчик Express, который обслуживает GET-запросы с пути `/boards`. В настоящее время этот обработчик просто записывает сообщение с помощью функции `serverLog` и возвращает полезную нагрузку JSON с единственным свойством `result`.

Как мы уже видели, мы должны реэкспортировать маршрутизатор из этого файла, как показано в последней строке предыдущего фрагмента кода. Затем мы можем зарегистрировать этот обработчик на нашем сервере Express в файле `express/main.ts`:

```
...существующий код
app.use('/', userRoutes.router);
app.use('/api', dataRoutes.router);
...существующий код
```

Мы добавили новый вызов функции `app.use` для регистрации всех маршрутов из файла `dataRoutes.ts` на сервере Express по базовому URL-пути `/api`. Это означает, что для доступа к любой из наших конечных точек REST нам потребуется добавить путь `/api` в начало URL-адреса. Использование базового URL-адреса таким образом позволяет нам отличать стандартные обработчики Express от обработчиков REST. Теперь давайте подключимся к нашей базе данных в обработчике `/boards` и вернем все найденные строки:

```
router.get(`/boards`, async (req: any, res: any, next: any) => {
   serverLog(`GET /boards`);
   let db = new Database('./database/board_sales_db.db');

   let boardsArray: any[] = [];

   db.each(`select
      b.id, b.name, b.short_description,
      b.long_description, b.img
      from Board b

   `, (err: Error, row: any) => {
      let board = {
         id: row.id, name: row.name,
         short_description: row.short_description,
         long_description: row.long_description,
         img: row.img
      };

      boardsArray.push(board);
   }, (err: Error, count: number) => {
      // complete
      if (err) {
         serverLog(`err : ${err}`);
```

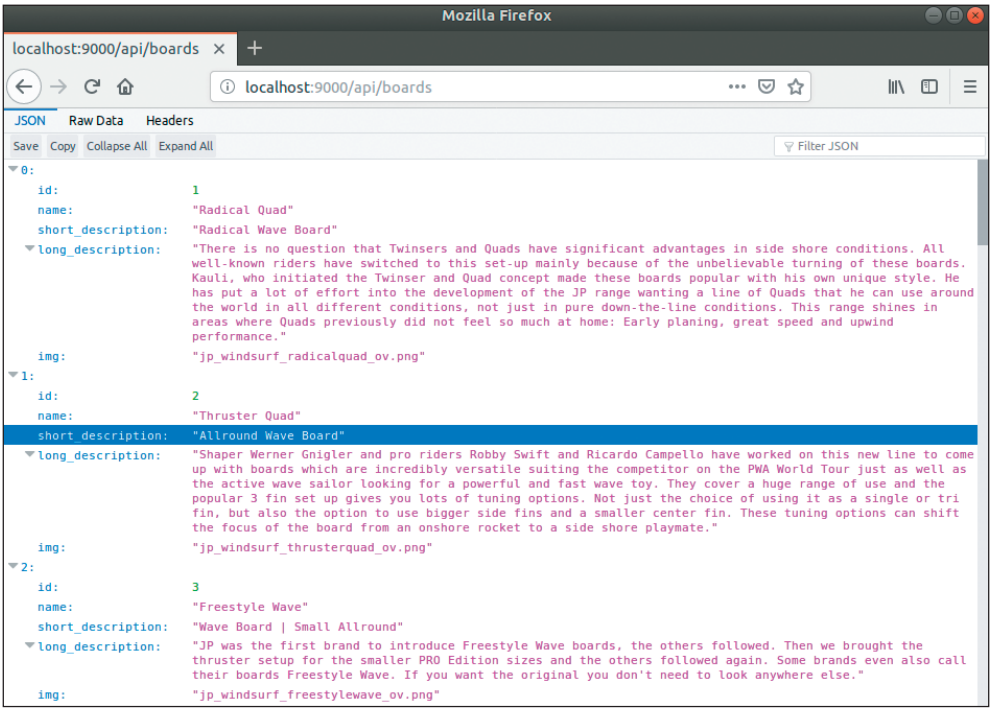
```
        res.status(503).send(err);
      } else {
        console.log(` => returning array`);
        res.json(boardsArray);
      }
    });
});
```

Здесь мы приступили к использованию API Sqlite3 для подключения к базе данных и возврата результатов. Мы обновили обработчик, чтобы создать локальную переменную `db`, которая является новым экземпляром класса `Database`. Эта переменная представляет соединение с базой данных Sqlite3 и имеет единственный аргумент конструктора, который является путем к самому файлу базы данных Sqlite3. Затем наш обработчик создает новый массив с именем `boardsArray`, который устанавливается как пустой массив.

После этого мы используем функцию `each` класса `Database` для выполнения SQL-запроса. У этой функции три параметра. Первый параметр типа `string` является SQL-командой для выполнения. Второй параметр – это функция, которая будет выполняться для каждого результата, возвращаемого базой данных, а третий параметр – функция, которая будет выполнена после того, как будут возвращены все результаты. API Sqlite3, с которым мы работаем, использует стандартный механизм обратного вызова. Другими словами, вторым параметром, который мы определили при вызове `db.each`, является функция обратного вызова, которая будет вызываться для каждой записи, возвращаемой базой данных. Третий параметр – это дополнительная функция обратного вызова, которая будет вызываться после прочтения всех записей.

Этот прием двойного обратного вызова на самом деле очень удобен. В рамках первого обратного вызова мы знаем, что имеем дело только с одной записью базы данных. Поэтому мы можем создать объект JavaScript, который представляет эту запись, и добавить его в наш массив записей для возврата. В рамках второго обратного вызова мы знаем, что все записи были обработаны из базы данных. Если произошла какая-либо ошибка, мы возвращаем ответ со статусом `503` и приводим сообщение об ошибке. Если все идет хорошо, мы возвращаем все содержимое локальной переменной `boardsArray` в виде структуры JSON.

При переходе по URL-адресу `localhost:9000/api/boards` теперь возвращается структура JSON с данными из нашей базы данных, как показано на приведенном ниже скриншоте:



Существует ряд конечных точек, которые нам понадобятся для предоставления необходимых данных для нашего приложения. Мы не будем подробно обсуждать каждую из них, поскольку реализация каждого обработчика маршрутов одинакова. Другими словами, подключиться к базе данных, выполнить SQL-запрос и вернуть результаты. Единственное различие между каждым из этих обработчиков – это маршрут обработчика, выполняемый SQL-запрос и соответствующая возвращаемая структура данных JSON. Для краткости мы перечислим только каждый из маршрутов конечной точки и используемый SQL-запрос. Пожалуйста, обратитесь к примеру кода, приведенному в этой главе, чтобы получить сведения о полной реализации.

Наш список производителей будет обслуживаться конечной точкой `/api/sources` и состоять из следующего SQL-запроса:

```
select id, name, logo from Manufacturer
```

Наш список типов досок будет обслуживаться конечной точкой `/api/board-types` и выполнять следующий SQL-запрос:

```
select id, name from BoardType
```

Привязка конкретной доски к производителю осуществляется с помощью конечной точки `/board-manufacturers`, которая выполнит следующий запрос:

```
select id, board_id, manufacturer_id from BoardManufacturer
```

Опять же, пожалуйста, обратитесь к примеру кода, чтобы увидеть полную реализацию.

Параметризованные конечные точки API

При проектировании конечных точек REST общее правило заключается в предоставлении конечной точки для каждой логической группы данных, которые будут запрашиваться. Имя этой конечной точки должно быть во множественном числе, что означает, что она вернет массив результатов. Таким образом, конечная точка `/api/boards` вернет список данных, относящихся ко всем доскам в базе данных. В этой конечной точке могут быть применены критерии поиска для фильтрации результатов. Например, URL-адрес `/api/boards?type=1&Manufacturer=2` должен применять значения типа и производителя в качестве фильтра, чтобы ограничить поиск всех досок в системе.

Когда нам нужно работать с одним экземпляром элемента, мы добавляем идентификатор элемента, который хотим запросить, к имени конечной точки во множественном числе. Другими словами, чтобы получить информацию о доске с идентификатором 3, мы будем использовать URL-адрес `/api/boards/3`. Это соответствует нашей парадигме, когда вся информация, касающаяся досок в системе, извлекается из конечной точки `/api/boards`. Предоставляя идентификатор, мы ограничиваем количество досок, возвращаемое этой конечной точкой, одной доской.

Помните, что мы можем предоставить несколько конечных точек, которые будут извлекать одни и те же данные. Рассмотрим конечную точку, доступ к которой осуществляется через URL-адрес `/api/boardtypes/2/Board`. Эта точка вернет один тип доски с идентификатором 2, а затем вернет все доски, связанные с этим типом. Аналогично, у нас может быть конечная точка с именем `/api/boards/1/board-types`, которая возвращает типы досок, связанные с доской с идентификатором 1. Всегда целесообразно проектировать конечные точки API, думая о том, как пользователи будут взаимодействовать с нашими данными и помогать им работать с ними интуитивно понятным способом.

Чтобы разрешить конечной точке Express использовать параметр в имени конечной точки, мы используем двоеточие и предоставляем имя для этого параметра:

```
router.get(`/boards/:boardId`, (req: any, res: any, next: any) => {
   let boardId = req.params.boardId;
   serverLog(`GET /boards/${boardId}/sizes`);
   let db = new Database(databaseName);
   let board: any = {};

   let sqlString = `select b.id, b.name, b.short_description,
```

```
        b.long_description, b.img
        from Board b
        where b.id=${boardId}`;

  ... db.each code ...
```

Здесь мы определили обработчик с маршрутом `/boards/:boardId`. Поэтому Express будет сопоставлять этот обработчик с любым запросом, который начинается с `/boards/` и добавляет параметр. Это означает, что обработчик будет вызываться, если входящий URL-адрес совпадает с адресами `/boards/1`, или `/boards/2`, или даже `/boards/abcd-efg`. Параметр `BoardId` доступен через свойство `req.params`, используя имя самого параметра. В этом случае мы назвали параметр `boardId`, и поэтому он доступен через свойство `req.params.boardId`. Обратите внимание, что выполняемый нами оператор SQL был обновлен для использования клаузы where, включающей этот параметр.

Мы не ограничены одним параметром в обработчике маршрута, и этот параметр также может присутствовать в любом месте нашего URL-адреса. В качестве примера рассмотрим следующий обработчик:

```
router.get(`/boards/:boardId/sizes`,
  (req: any, res: any, next: any) => {

  let boardId = req.params.boardId;
  serverLog(`GET /boards/${boardId}/sizes`);
  let db = new Database(databaseName);

  let boardSizeArray: any[] = [];

  let sqlString = `select board_id, volume, length, width,
    sail_min, sail_max
    from BoardSize where board_id=${boardId}`;

... db.each code ...
```

Здесь у нас есть обработчик, определенный как `/boards/:boardId/sizes`. Этот обработчик будет вызываться URL-адресом, таким как `/boards/1/sizes` или `/board/3/sizes`. Обратите внимание, что независимо от того, где указан параметр `boardId` в адресе, он по-прежнему доступен из свойства `req.params.boardId`. В предыдущем примере мы использовали параметр `boardId` для фильтрации записей из таблицы `BoardSize` на основе `boardId`.

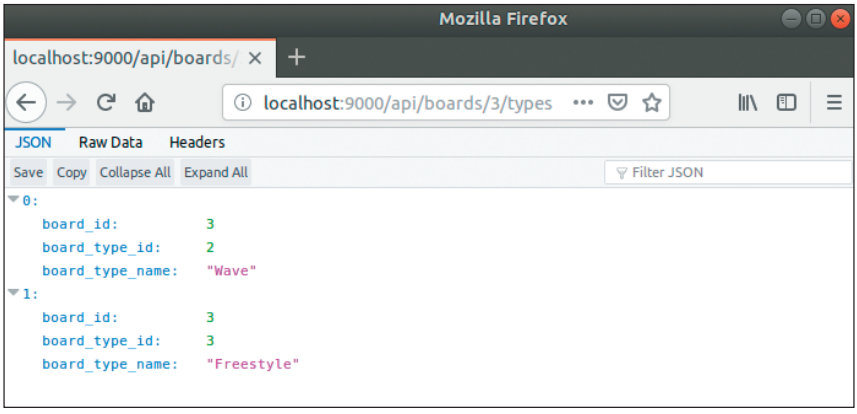
Аналогичным образом мы можем запросить список типов, которые есть у конкретной доски, используя следующий обработчик:

```
router.get(`/boards/:boardId/types`,
  (req: any, res: any, next: any) => {
    let boardId = req.params.boardId;
```

```
        serverLog(`GET /boards/${boardId}/types`);
        let db = new Database(databaseName);

        let boardSizeArray: any[] = [];

        let sqlString = `select b.id, bbt.board_type_id,
        bt.name from Board b
        INNER JOIN BoardBoardType bbt ON b.id = bbt.board_id
        INNER JOIN BoardType bt ON bbt.board_type_id = bt.id
        where b.id = ${boardId}`;

    ... db.each code ...
```

Здесь мы определили обработчик, который будет вызываться URL-адресом `/boards/:boardId/types`. Этот обработчик вернет все типы досок, которые связаны с конкретной доской. Обратите внимание, что этот обработчик использует два оператора `INNER JOIN` для извлечения данных из таблицы `BoardBoardType`, а также из таблицы `BoardType`.

Возвращенная структура JSON выглядит так:



Здесь можно увидеть результаты запроса REST API к конечной точке `/api/boards/3/types`. Наши операторы `INNER JOIN` совпали и вернули значения из таблицы `BoardBoardType` (которая связывает доску с типом доски), а также имена из `BoardTable`, которые будут использоваться для отображения.

Давайте перечислим конечные точки API, которые мы создали к настоящему моменту:

- `/api/boards` вернет массив всех досок, которые есть в системе;
- `/api/manufacturers` вернет массив всех производителей в системе;
- `/api/board-manufacturers` вернет массив всех досок и их производителей;
- `/api/boards/:boardId` вернет информацию, относящуюся к одной доске в системе, идентифицируемую определенным `boardId`;

- `/api/boards/:boardId/sizes` вернет массив доступных размеров для конкретной доски;
- `/api/boards/:boardId/types` вернет массив связанных типов для конкретной платы.

Обратите внимание, что у конкретной доски есть только один производитель, и нам потребуется информация, относящаяся к производителю (например, его название и логотип), когда мы показываем доску. По этой причине мы обновим SQL-запрос для `/api/boards` и `/api/boards/:boardId`:

```
select b.id, b.name, b.short_description,
   b.long_description, b.img,
   bm.manufacturer_id as mfid,
   mf.name as mfname,
   mf.logo as mflogo
   from Board b
   INNER JOIN BoardManufacturer bm
   ON b.id = bm.board_id
   INNER JOIN Manufacturer mf
   ON bm.manufacturer_id = mf.id
```

Здесь мы выбираем записи из таблицы `Board`, а затем соединяем эти данные с таблицами `BoardManufacturer` и `Manufacturer`. Таблица `BoardManufacturer` связывает конкретную доску с производителем, а таблица `Manufacturer` содержит название и логотип, которые будут использоваться. Мы также можем разом вернуть все эти данные.

Службы REST в Angular

Чтобы взаимодействовать с нашими недавно созданными конечными точками REST, мы будем следовать парадигме проектирования «Единственная ответственность» и создадим службу Angular. Она будет помечена декоратором `@Injectable`, который позволяет нам использовать фреймворк внедрения зависимости Angular для внедрения экземпляра этой службы, когда и где это необходимо. Давайте создадим новый файл в каталоге `/app/services` с именем `board.service.ts`:

```
import {Injectable} from "@angular/core";
import {HttpClient} from "@angular/common/http";
import {Observable} from "rxjs";
@Injectable({
   providedIn: 'root'
})
export class BoardService {
   constructor(private httpClient: HttpClient) { }
   getManufacturerList(): Observable<Object> {
```

```
        return this.httpClient.get('/api/manufacturers');
    }
    getBoardTypesList(): Observable<Object> {
        return this.httpClient.get('/api/board-types');
    }
    getBoardsList(): Observable<Object> {
        return this.httpClient.get('/api/boards');
    }
    getBoardManufacturers(): Observable<Object> {
        return this.httpClient.get(‘/api/board-manufacturers’);
    }
    getBoardDetails(boardId: number): Observable<Object> {
        return this.httpClient.get('/api/boards/${boardId}');
    }
    getBoardSizes(boardId: number): Observable<Object> {
        return this.httpClient.get('/api/boards/${boardId}/sizes');
    }
    getBoardTypes(boardId: number): Observable<Object> {
    return this.httpClient.get('/api/boards/${boardId}/types');
    }
}
```

Здесь мы начинаем с импорта декоратора `Injectable` и класса `HttpClient`, которые мы будем использовать для асинхронных запросов, к нашей конечной точке REST. Каждый вызов с использованием `HttpClient` будет возвращать объект Observable, поэтому нам нужно импортировать и его.

У класса `BoardService` есть конструктор, который использует фреймворк внедрения зависимости Angular для объявления закрытой переменной с именем `httpClient`, через которую мы можем вызывать наш API. У него также имеется ряд функций, каждая из которых относится к нашим конечным точкам REST, как обсуждалось ранее. Обратите внимание, что для последних трех функций, `getBoardDetails`, `getBoardSizes` и `getBoardTypes`, требуется один параметр с именем `boardId` типа `number`. Этот параметр затем вставляется в URL-адрес. Это то, что мы называем правильной параметризованной конечной точкой REST.

Создание службы таким способом и предоставление функций для вызова наших конечных точек означает, что мы скрываем тонкости конструирования URL-адреса от нашего вызывающего кода. Чтобы получить доступные типы досок, мы просто вызываем `getBoardTypes` и предоставляем идентификатор доски в качестве единственного аргумента. Это гораздо проще, чем помнить, что URL-адрес – это `/api/boards/:boardId/types`. Это также делает наш код более пригодным для повторного использования и гарантирует, что единственное место, где нам нужно понять, как создать правильный URL-адрес, находится в пределах самого класса `BoardService`.

Обратите внимание, что в этой версии каждый вызов функции возвращает тип `Observable <Object>`. Хотя это синтаксически правильно и мы фактически возвращаем структуру JSON, которая отображается в тип `Object`, для нашего кода было бы намного лучше определить интерфейсы, которые возвращает каждый вызов.

Таким образом, любой пользователь службы сразу поймет, какая информация доступна для каждого вызова функции.

Давайте обновим функцию `getManufacturersList`, чтобы использовать интерфейс следующим образом:

```
export interface INameId {
   id: number;
   name: string;
}

export interface IManufacturer extends INameId {
   logo: string;
}

... существующий код

getManufacturerList(): Observable<IManufacturer[]> {
   return this.httpClient.get('/api/manufacturers')
      as Observable<IManufacturer[]>;
}
```

Здесь мы определили три интерфейса. Интерфейс `INameId` состоит из идентификатора типа `number` и имени типа `string`. Интерфейс `IManufacturer` добавляет свойство `logo` к интерфейсу `INameId`. Мы также обновили функцию `getManufacturerList`, чтобы вернуть объект `Observable` типа `IManufacturer[]`. Этот интерфейс соответствует структуре данных JSON, которая возвращается из нашей конечной точки REST.

Теперь мы можем определить три дополнительных интерфейса в качестве завершающего штриха:

```
export interface IBoardSize {
   board_id: number;
   volume: number;
   width: number;
   length: number;
   sail_min: string;
   sail_max: string;
}
```

```
export interface IBoardType {
   board_id: number;
   board_type_id: number;
   board_type_name: string;
}

export interface IBoard extends INameId {
   short_description: string;
   long_description: string;
   img: string;
   manufacturer_logo: string;
   mfid: number;
   mfname: string;
   mflogo: string;
}
```

Здесь мы определили три интерфейса: `IBoardSize`, `IBoardType` и `IBoard`. Каждое свойство этих интерфейсов соответствует структуре JSON, которую мы возвращаем из наших конечных точек REST.

Спецификация OpenAPI

В крупных системах определение REST API и данные, возвращаемые каждой конечной точкой, необходимо использовать по-разному. Возьмите для примера документацию. Общедоступному API REST необходимо, чтобы каждая из его конечных точек была указана в формальном документе, и этот документ еще должен определять тип каждого свойства, а также ограничения. В качестве примера этих ограничений рассмотрим POST-вызов к конечной точке. Если наша точка определяет свойство типа number, то как пользователь этой конечной точки узнает, каковы верхние и нижние пределы этого числа? Это 32- или 64-разрядное число? Оно подписано или нет? Если конечная точка определяет свойство `string`, то сколько символов разрешено?

Спецификация OpenAPI – это стандартизированный способ описания природы REST API.

Наряду с описанием доступных конечных точек этот стандарт также позволяет описывать, какие операции разрешены для каждой конечной точки. Разрешает ли конечная точка POST и PUT, или это просто GET-запросы? Если это GET-запрос, то какие параметры запроса разрешены? Спецификация OpenAPI позволяет производителю REST API сделать все нюансы конкретной системы доступными в стандартном формате.

Благодаря спецификации OpenAPI, доступной для REST API, мы можем использовать ряд инструментов, помогающих применять этот API, включая документацию, генераторы клиентского кода и даже автоматическое создание заглушек

сервера. Мы не будем здесь обсуждать генерирование спецификации OpenAPI, но имейте в виду, что это стандарт, и создание спецификации для вашего API поможет тем, кто потребляет ее в огромной степени.

Одним из преимуществ спецификации OpenAPI является то, что мы можем создавать клиентские библиотеки на основе определений API. Это означает, что интерфейсы, которые мы только что создали вручную, могут автоматически генерироваться инструментами. Если мы включим эти инструменты в наши этапы сборки, то можем быть уверены, что любые изменения в API будут автоматически отражены в нашем коде. Кроме того, если API имеет критические изменения, наша сборка не будет выполнена, если она полагается на свойство, которое внезапно исчезло. Такие ошибки при обнаружении их на ранних этапах процесса разработки гораздо дешевле исправить, чем искать их в тестовой или производственной среде.

Приложение BoardSales

Нашему приложению потребуется ряд обновлений, чтобы использовать API REST, который мы определили, и сгенерировать нужные нам страницы. Шаги, которые нам нужно выполнить, в целом таковы:

- создать компонент списка досок, чтобы вывести список всех досок, полученных из конечной точки `/api/boards`;
- интегрировать этот компонент в нашу главную страницу;
- загрузить и отобразить список производителей и список типов досок в нашем компоненте `sidenav` из конечных точек `/api/manufacturers` и `/api/board-types`;
- ответить на событие щелчка на доске в списке досок и отобразить информацию о доске в нашем компоненте `rightscreen`;
- реагировать на события фильтрации, когда пользователь фильтрует список досок на основе производителя или типа.

Внося изменения в приложение, мы обсудим использование шаблона проектирования Domain Events, а также подробно рассмотрим различные способы применения Observables для работы с данными и манипулирования ими.

Компонент BoardList

Давайте начнем обновление нашего приложения с создания и интеграции компонента `board-list`, который будет отображать список всех доступных досок. Этот компонент будет запрашивать данные у конечной точки `/api/boards` и служить главной страницей нашего сайта. BoardListComponent можно создать с помощью интерфейса командной строки Angular:

```
ng generate component board-list --skip-import
```

С помощью этой команды будет сгенерирован каталог /src/app/board-list, а также созданы файлы .ts, .html, .css и .spec.ts для нашего компонента. Нам нужно зарегистрировать thisBoardListComponent в качестве модуля в файле app.module.ts:

```
@NgModule({
   declarations: [
      ...имеющиеся компоненты...
      BoardListComponent
   ],
   imports: [
      ...существующий код...
```

Здесь мы просто добавили класс BoardListComponent в список компонентов в массиве declarations. Помните, что добавление записи в данном случае позволит нам включить элемент <app-board-list> в HTML-файл нашего приложения. Теперь мы можем обновить HTML-файл SecureComponent:

```
<app-navbar></app-navbar>
<app-sidenav></app-sidenav>

<div id="sidenav_expand_panel" class="sidenav_expand"
   (click)="showHideSideClicked()">
   <span id="show-hide-side-button" class="filter-button fa">
   </span>
</div>

<app-rightscreen (notify)='onNotifyRightWindow($event)'>
</app-rightscreen>

<div id="main" class="main-content-panel">
   <div> ... page title ... </div>
   <div class="main-content">
      <app-board-list></app-board-list>
   </div>
</div>
```

Здесь у нас есть четыре основных компонента, которые составляют нашу главную страницу, к каждому из которых обращаются их имена элементов. Элемент <app-navbar> будет отображать панель навигации в верхней части экрана, а элемент <app-sidenav> – боковую панель слева, которую мы будем использовать для фильтрации. Элемент <app-rightscreen> будет отображать панель сведений при выборе доски. Затем у нас идет элемент , представляющий панель основного содержимого нашей страницы, и внутри него элемент <app-board-list>, который будет отображать полный список досок.

Обратите внимание, что предыдущий пример HTML-кода представляет собой урезанную версию полного файла, чтобы показать размещение основных эле-

ментов. Элементы `...page title...` здесь не показаны. Эти HTML-файлы сопровождаются соответствующими CSS-стилями, которые используются для каждого элемента и хранятся в файлах `.css`. Опять же, ради краткости мы не будем обсуждать эти стили или перечислять их здесь, поэтому, пожалуйста, обратитесь к примеру кода, чтобы получить сведения о полной реализации.

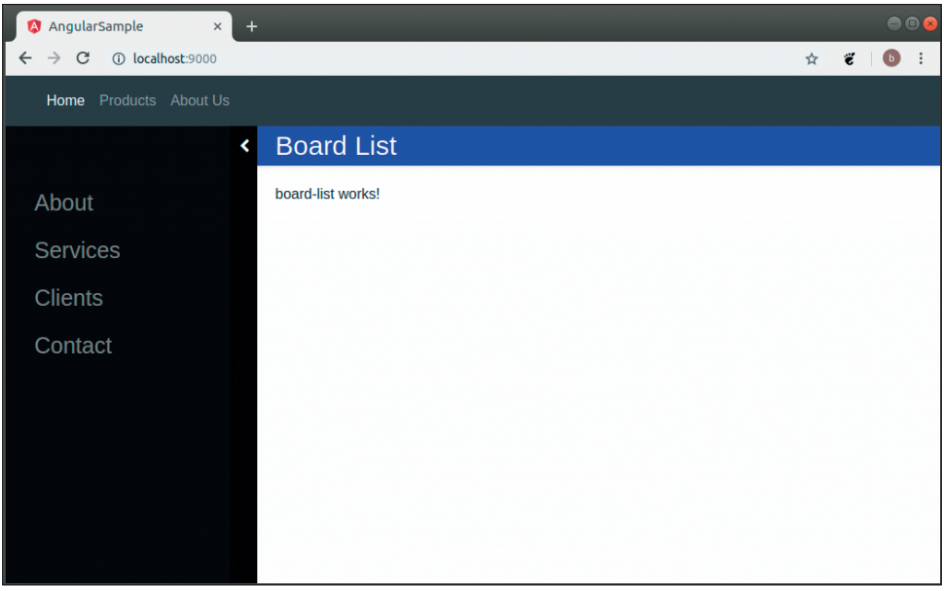
Мы также ввели `<div>`-элемент с идентификатором `sidenav_expand_panel` между элементами `<app-sidenav>` и `<app-rightscreen>`. Это обновленная версия кнопки, которую мы использовали ранее для развертывания и свертывания элемента `<app-sidenav>`, и на самом деле это панель шириной `30px`, которая заполняет страницу сверху вниз. Чтобы анимировать эту панель и компенсировать дополнительную ширину, нам нужно обновить `SecureComponent`:

```
showNavPanel() {
   this.sideNav.showNav();
   document.getElementById('main').style.marginLeft = "280px";
   document.getElementById('sidenav_expand_panel')
      .style.left = "250px";
}
hideNavPanel() {
   this.sideNav.closeNav();
   document.getElementById('main').style.marginLeft = "30px";
   document.getElementById('sidenav_expand_panel')
      .style.left = "0px";
}
showDetailPanel() {
   this.rightScreen.openRightWindow();
   document.getElementById('main').style.transform =
      "translateX(-100%)";
   document.getElementById('sidenav_expand_panel').style.
      transform = "translateX(-100%)";
}
hideDetailPanel() {
   this.rightScreen.closeRightWindow();
   document.getElementById('main').style.transform =
      "translateX(0%)";
   document.getElementById('sidenav_expand_panel').style.
      transform = "translateX(0%)";
}
```

Здесь мы обновили функцию `showNavPanel`, включив в нее свойство элемента `style.left` с идентификатором `sidenave_expand_panel`, равным `250px`. Это очень похоже на наш существующий код, который устанавливает свойство `style.marginLeft` для элемента `main`. Аналогично, в функции `hideNavPanel` мы устанавливаем свойство `style.left` для `sidenav_expand_panel` равным `0px`. `ShowDetailPanel` и `hideDetailPanels` также были обновлены, чтобы

включить свойство `style.transform` для `sidenav_expand_panel`. Опять же, этот код похож на код анимации элемента `main`.

Теперь на нашей главной странице должен отображаться компонент `board-list`:



Визуализация данных REST

Теперь, когда у нас есть основа `BoardListComponent`, мы можем интегрироваться с `BoardService` и запросить конечную точку REST `/api/board` для получения списка отображаемых досок. Наши обновления выглядят так:

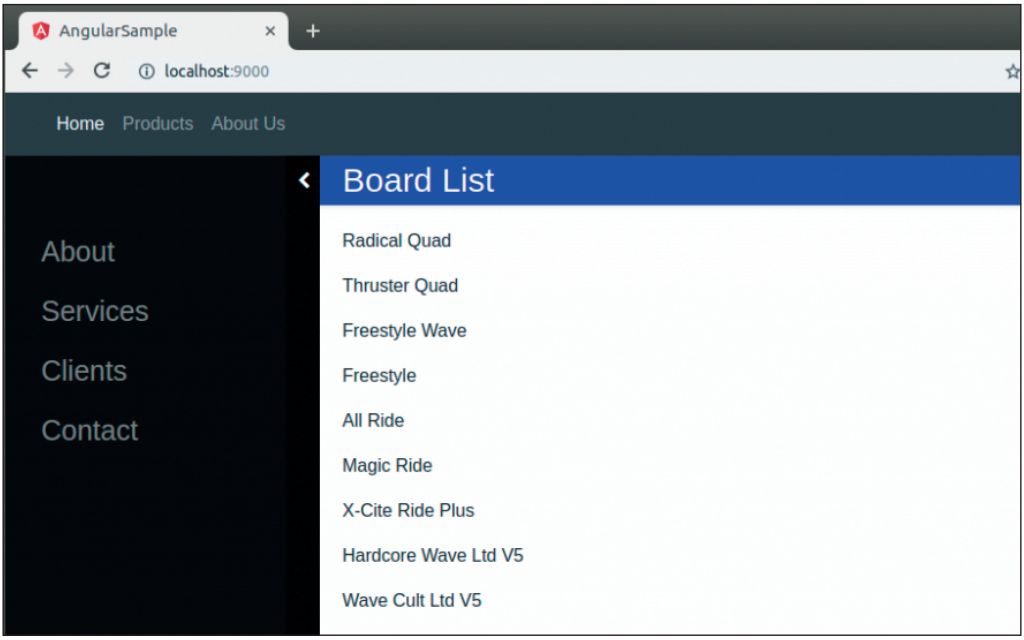
```
export class BoardListComponent implements OnInit {
  boardList: IBoard[] = [];
  constructor(private boardService: BoardService) { }
  ngOnInit() {
    this.boardService.getBoardsList()
    .subscribe((result: IBoard[]) => {
      this.boardList = result;
    });
  }
}
```

Здесь у нас есть свойство `boardList`, которое содержит массив элементов типа `IBoard`. Затем мы обновили нашу функцию-конструктор, чтобы внедрить экземпляр службы `BoardService` и сохранить его в закрытой перемен-

ной с именем `boardService`. После этого функция `ngOnInit` вызывает функцию `getBoardList` для экземпляра `boardService` и определяет функцию `subscribe`. Внутри этой функции мы просто устанавливаем в качестве значения локальной переменной `boardList` результат вызова конечной точки REST. Мы можем просто обновить наш HTML-файл следующим образом:

```
<div *ngFor="let board of boardList;">
  <p>{{board.name}}</p>
</div>
```

Здесь мы обновили HTML-шаблон и добавили элемент `<div>`, который использует синтаксис `*ngFor` для перебора всех элементов в массиве `boardList`. Каждый элемент массива будет генерировать элемент `<p>` и отображать свойство `name` в DOM. На этом этапе, после нескольких строк кода, можно увидеть результаты запроса нашей конечной точки REST:



Теперь, когда у нас есть основа этой страницы, мы можем обновить HTML-код, чтобы создать более визуально привлекательный список, который будет включать в себя изображение доски, логотип производителя и краткое описание. HTML-код для каждой доски будет состоять из двух строк: в верхней строке будет отображаться название доски и краткое описание, а в нижней строке будет показано изображение и ее размеры. Наша верхняя строка выглядит так:

```
<div class="row">
  <div class="col-sm-8 board-name-panel">
```

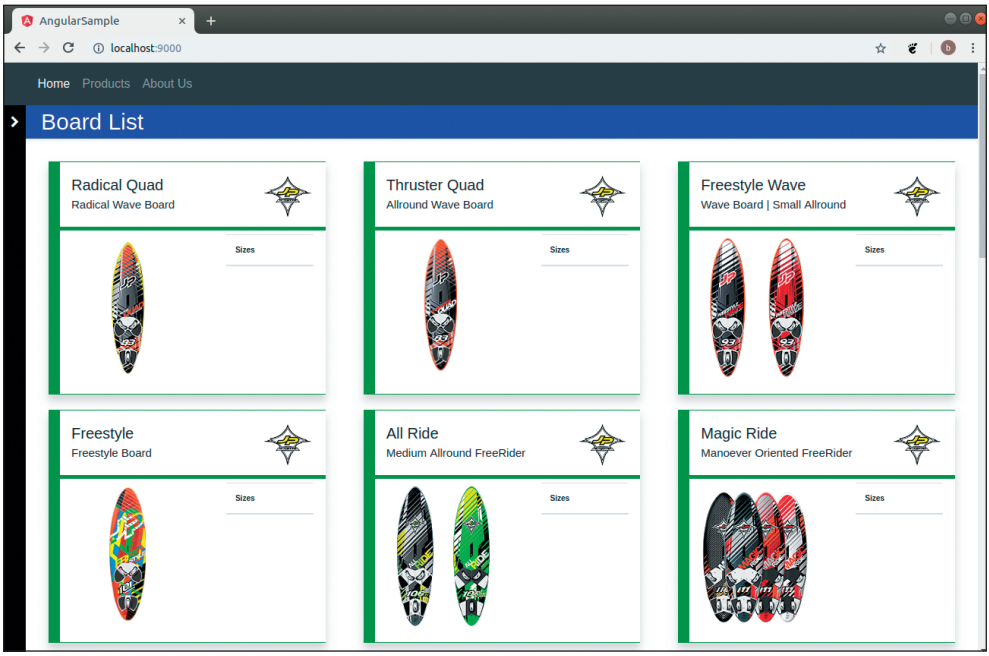
```
        <div class="board-name">{{board.name}}</div>
        <div class="board-desc">{{board.short_description}}</div>
      </div>
      <div class="col-sm-4 float-right board-name-panel">
        <img src="/assets/images/{{board.mflogo}}"
          class="img manufacturer-logo" />
      </div>
    </div>
```

Здесь мы определили строку `Bootstrap` и в ней две колонки размера `col-sm-8` и `col-sm-4`. Первая колонка используется для отображения свойств `board.name` и `board.short_description`, а вторая колонка – для отображения логотипа производителя посредством свойства `board.mflogo`.

Вторая строка в нашем HTML-коде для каждой доски выглядит так:

```
  <div class="row">
    <div class="col-sm-7">
      <div class="board-name">
        <img src="/assets/images/{{board.img}}"
          class="img board-image" />
      </div>
    </div>
    <div class="col-sm-5 size-table">
      <table class="table">
        <thead>
          <tr> <th scope="col">Sizes</th> </tr>
        </thead>
        <tbody>
          <div *ngFor="let size of board.sizes">
            <tr>
              <td>{{size.volume}}</td>
            </tr>
          </div>
        </tbody>
      </table>
    </div>
```

Здесь у нас есть элемент строки Bootstrap с двумя колонками размера `col-sm-7` и `col-sm-5`. Первая колонка используется для визуализации изображения доски, а вторая колонка содержит таблицу с заголовком `Sizes`. Этот элемент таблицы затем использует `*ngFor` для циклического перебора свойства `sizes` текущей доски, чтобы отобразить доступные размеры для каждой доски. Мы обсудим, как заполнять свойство `sizes`, немного позже, но с этим HTML-кодом и разумной порцией CSS-стилей наш список досок становится более привлекательным:



concatMap

В нашем списке досок в настоящее время используются данные из конечной точки REST `/api/boards`, и поэтому у него есть доступ к общей информации, касающейся доски, такой как `name`, `short_description` и логотип производителя. Помните, что большинство досок для виндсерфинга бывает разных размеров, причем у каждого размера бывает разный объем, ширина и длина, чтобы поддерживать различные размеры паруса. Таким образом, здесь мы имеем связь «один ко многим» между доской и ее размерами, так как у одной доски может быть много размеров.

Однако наши конечные точки REST не возвращают размеры доски в структуре JSON одной доски. Это означает, что для каждой доски в нашем списке нам нужно будет сделать дополнительный вызов к конечным точкам REST, чтобы получить ее размеры. Можно с легкостью утверждать, что конечная точка REST, которая извлекает информацию о доске, должна включать в себя массив, включая ее размеры. Возможно, это будет наиболее естественный способ поведения конечной точки.

Однако бывают случаи, когда для каждого элемента в коллекции необходимо выполнить вторичный вызов REST, поэтому мы будем использовать этот пример для обсуждения метода, использующего Observables, который выполняет это точное требование: оператор `concatMap`.

Оператор `concatMap` будет генерировать одно значение Observable за раз по порядку и разрешать подписку на значение этого объекта Observable. Он не будет испускать еще один элемент Observable, пока не завершится предыдущая подписка. Это означает, что мы можем обрабатывать элементы Observable один за другим и выполнить еще одно действие с элементом перед обработкой другого. Поэтому мы можем использовать оператор `concatMap` для вторичного вызова нашей конечной точки REST для каждого элемента в Observable. Помните, что конечная точка `/api/boards` возвращает массив досок. Нам нужно будет выполнить вызов конечной точки `/api/boards/:boardId/sizes` для каждого элемента массива, а затем обновить информацию об исходной доске, с которой мы работали, возвращенными размерами.

Нам нужно выполнить пять шагов, чтобы правильно использовать оператор `concatMap`, поэтому давайте внесем некоторые изменения в наш код и представим эти шаги один за другим.

Первый шаг – импорт правильных операторов из библиотеки RxJS:

```
import {from} from 'rxjs';
import {concatMap} from 'rxjs/operators';

interface IBoardExtended extends IBoard {
   sizes: IBoardSize[];
}

...определение класса...

ngOnInit() {
   this.boardService.getBoardsList()
      .subscribe((result: IBoardExtended[]) => {
      // this.boardList = result;
      // Здесь мы используем concatMap;
   });
}
```

Здесь мы начинаем с импорта функции `from` из библиотеки '`rxjs`', а также функции `concatMap` из библиотеки '`rxjs/operator`'. Затем мы определяем интерфейс с именем `IBoardExtended`, который добавляет свойство `sizes` к существующему определению интерфейса `IBoard`. Это свойство представляет собой массив типа `IBoardSize` и будет содержать результаты отдельных вызовов конечной точки `/api/board/:boardId/sizes`.

Затем мы обновили функцию `ngOnInit` и, в частности, закомментировали строку, которая устанавливает в качестве значения внутреннего свойства `boardList` результат вызова функции `getBoardList`. Мы будем использовать переменную `result` с оператором `concatMap`. Однако, прежде чем мы используем этот оператор, нам потребуется создать Observable из массива `IBoardExtended`:

```
let boardSizes = from(<IBoard[]>result).pipe(
   ...  Здесь мы передаем каждый элемент массива
);
```

Здесь мы создаем локальную переменную `boardSizes`, которая будет содержать результаты вызова функции `pipe`. Мы используем функцию from для создания Observable из массива досок, которые были получены в результате запроса конечной точки REST в `/api/boards`. Затем мы вызываем функцию `pipe` для этого элемента Observable. Этот код говорит о том, что мы создаем Observable из массива `IBoard`, а затем передаем эти результаты в другую функцию.

Внутри функции pipe мы можем использовать функцию `concatMap`:

```
let boardSizes = from(<IBoard[]>result).pipe(
   concatMap (
      // Первая функция;
      (board: IBoardExtended) => {
         // Вызывается для каждой доски в массиве;
         // Возвращает Observable;
      },
      // Вторая функция;
      (board: IBoardExtended, sizes: IBoardSize[]) => {
         // board - это исходный элемент массива;
         // sizes - результат Observable;
      }
   ) // Завершаем вызов concatMap;
);
```

Здесь мы вызываем функцию `concatMap` внутри функции `pipe`. Поэтому мы добавляем наш элемент Observable в функцию `concatMap`.

Функция `concatMap` получает один элемент Observable и может затем вызвать еще одну функцию до получения следующего элемента Observable. Функция `concatMap` может использоваться с одним параметром или двумя. Первый параметр, который должен быть функцией, – это функция, которая будет вызываться для каждого элемента в Observable. Второй параметр, который также является функцией, вызывается после завершения первой функции и предоставляет два аргумента. Первый аргумент – это значение исходного Observable, которое использовалось, а второй аргумент – это результат первой функции. Другими словами, первая функция используется для работы с элементом Observable, а вторая функция является результатом этой операции. Это немного легче понять с помощью примера:

```
concatMap (
   // Первая функция;
   (board: IBoardExtended) => {
      console.log(`concatmap 1 : ${board.name}`);
```

```
        return this.boardService.getBoardSizes(board.id);
      },
      // Вторая функция;;
      (board: IBoardExtended, sizes: IBoardSize[]) => {
        console.log(`concatmap 1 : board.id : ${board.id}`);
        console.log(`concatmap 1 : sizes.length :
          ${sizes.length}`);
        board.sizes = sizes;
        return result;
  })
```

Здесь мы определили обе функции как параметры самой функции `concatMap`. Первая функция записывает свойство элемента массива `name`, который был передан через аргумент `board`. Затем она вызывает функцию `getBoardSizes`, используя идентификатор входящего элемента `board`. Эта функция будет вызываться для каждого элемента массива. Функция `getBoardSizes` выполнит вызов REST и возвратит Observable, который будет содержать результат вызова API REST.

Вторая функция записывает в консоль два сообщения, которые показывают нам идентификатор доски, с которой мы в настоящее время имеем дело, и длину массива `sizes`, который был получен из конечной точки `/api/boards/:boardId/sizes`. Затем она присваивает массив `sizes` свойству `sizes` исходного элемента массива `board` и возвращает значение исходного аргумента `result`.

Осталось только подписаться на `concatMap` и установить внутреннее свойство `boardList`:

```
  this.boardService.getBoardsList()
     .subscribe((result: IBoardExtended[]) => {
     // this.boardList = result;
     let boardSizes = from(<IBoard[]>result).pipe(
        ... функции concatMap
     );

     boardSizes.subscribe((boardList: IBoardExtended[]) => {
        this.boardList = boardList;
     });
  });
```

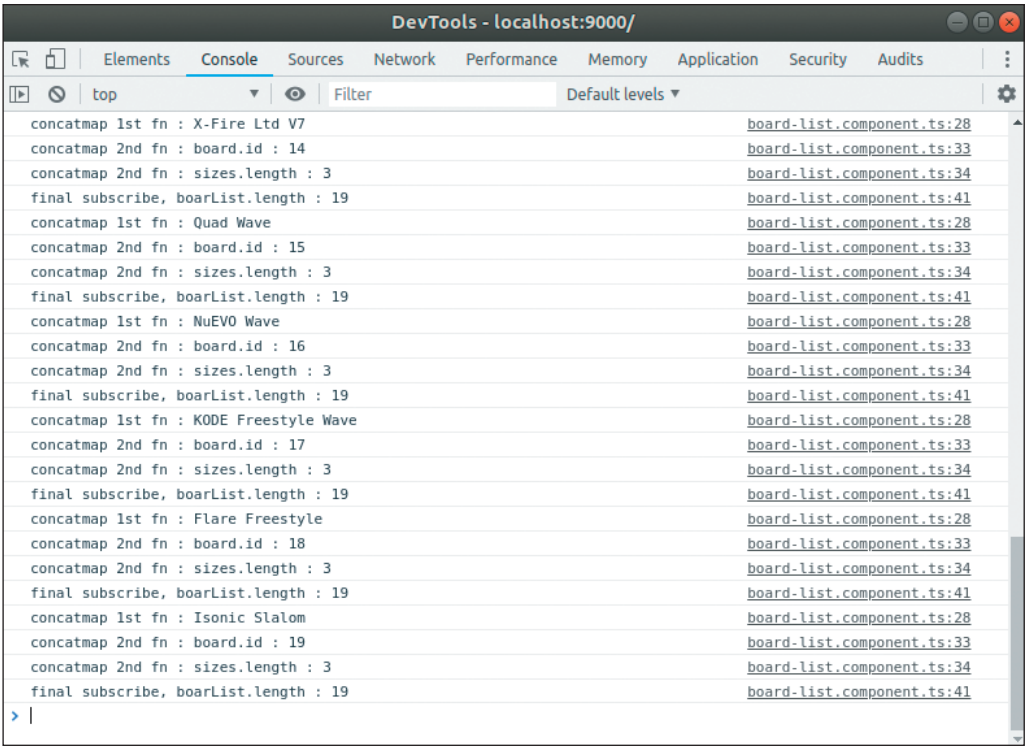
Здесь мы вызываем функцию `subscribe` для переменной `boardSizes` и в рамках этой анонимной функции устанавливаем в качестве значения внутреннего свойства `boardList` результат Observable.

Итак, чего мы достигли? Вот шаги, которые были выполнены в этом теле кода.

1. Подписаться на вызов конечной точки `/api/boards`.
2. Создать элемент Observable из возвращаемого массива из переменной `result`.

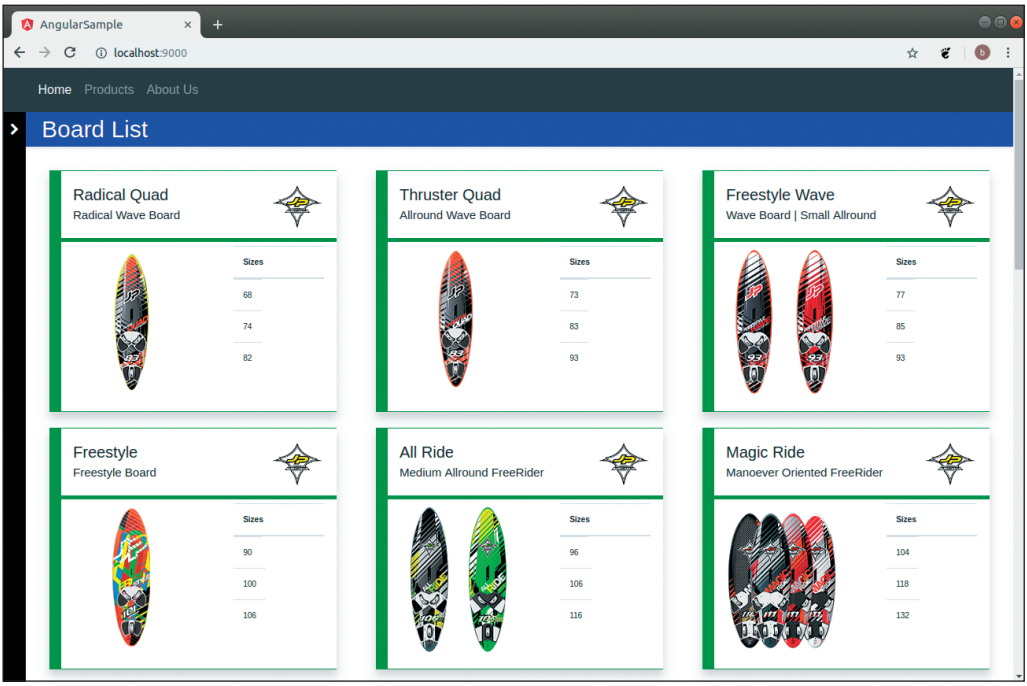
3. Передать элемент Observable в функцию concatMap.
4. Определить первую функцию, которая будет вызываться для каждого элемента массива.
5. В рамках этой функции вызвать конечную точку `/api/boards/:boardId/sizes`.
6. Когда этот вызов REST завершится, выполнить вторую функцию, предоставленную для `concatMap`.
7. Присвоить возвращенный массив из второго вызова REST свойству `sizes` исходного элемента массива.
8. Подписаться на Observable, созданный с помощью вызовов `from` и `pipe`.
9. Присвоить обновленный массив внутреннему свойству `boardList`.

Если мы посмотрим на журналы консоли, которые создаются в этом теле кода, то увидим следующее:



Здесь наши журналы показывают, что функция `concatMap` действительно обрабатывает каждый элемент массива один за другим, как и ожидалось.

После этого у каждой доски теперь будет свойство `sizes`, и наш HTML-код сможет отображать эти размеры в таблице Sizes:



forkJoin

Наша работа с экраном Board List еще не завершена. Наряду с набором размеров, связанных с доской, каждая доска также имеет набор типов. Помните, что доска может быть классифицирована как доска для волн, доска для фристайла, доска для слалома или как комбинация всех трех типов. Это вводит еще одну связь «один ко многим» между доской и ее типами наряду с существующей связью «один ко многим» между доской и ее размерами. У нас уже есть конечная точка REST, которая будет возвращать типы для каждой доски и поэтому может копировать функциональность concatMap для получения списка типов досок аналогичным образом, как показано ниже:

```
this.boardService.getBoardsList()
   .subscribe((result: IBoardExtended[]) => {
   let boardSizes = from(<IBoard[]>result).pipe(
      ...код concatMap ...
   );
   let boardListTypes = from(<IBoard[]>result).pipe(
      concatMap ((board: IBoardExtended) => {
         return this.boardService.getBoardTypes(board.id);
      }, (board: IBoardExtended, types: IBoardType[]) => {
         board.types = types;
```

```
        return result;
      })
);
```

Здесь мы используем тот же базовый код для определения переменной `boardListTypes`, которая будет содержать результат вызова функции `concatMap`. Однако в этом варианте вызывается функция `getBoardTypes`, содержащаяся в `listService`, и присваивается результат свойству доски `types`. Нам нужно будет обновить определение интерфейса `IBoardExtended` следующим образом:

```
interface IBoardExtended extends IBoard {
  sizes: IBoardSize[];
  types: IBoardType[];
}
```

Здесь мы просто добавили свойство типов в интерфейс `IBoardExtended`, который будет содержать массив элементов `IBoardType`. Вторая анонимная функция, переданная в функцию `concatMap`, устанавливает это свойство, когда конечная точка REST возвращает результат.

Если мы продолжим в том же духе, что и предыдущий код `concatMap`, нам нужно будет также подписаться на `boardListTypes`, как показано ниже:

```
boardSizes.subscribe((boardList: IBoardExtended[]) => {
  this.boardList = boardList;
});
boardListTypes.subscribe((boardList: IBoardExtended[]) => {
  this.boardList = boardList;
});
```

Здесь мы подписываемся как на `boardSizes`, так и на `boardListTypes` и устанавливаем в качестве значения внутреннего свойства `boardList` возвращаемый массив.

Однако тут у нас есть потенциальная проблема.

Из-за асинхронного характера вызова конечных точек REST и асинхронного характера работы с Observables мы не уверены, в каком порядке будут выполняться конечные функции `subscribe`. Может быть, что `boardSizes` завершится первым, а затем через несколько миллисекунд завершится `boardListTypes`. Или может быть наоборот. Если у нас есть код, который требует завершения обоих элементов Observables, прежде чем продолжить, мы не можем определить, что они оба завершены. К счастью, библиотека RxJS предоставляет механизм ожидания завершения нескольких Observables. Это функция `forkJoin`.

Функция `forkJoin` позволяет подписаться на массив Observables и будет выполняться только после завершения всех Observables, как показано ниже:

```
forkJoin ([boardSizes, boardListTypes]).subscribe((result) => {
  console.log(`forkJoin : result.length : ${result.length}`);
  console.log(`forkJoin : result[0].length :
    ${result[0].length}`);
  console.log(`forkJoin : result[1].length :
    ${result[1].length}`);
  this.boardList = result[1];
});
```

Здесь мы вызываем функцию `forkJoin`, которая использует единственный параметр. Этот параметр представляет собой массив всех Observables, которые нам нужно ждать. Как обычно, функция `forkJoin` генерирует новый Observable, поэтому нам нужно подписаться на него, а затем обработать результат. Аргумент `result` в функции `subscribe` – это, по сути, массив всех результатов каждого исходного Observable. Это означает, что длина массива `result` будет соответствовать количеству Observables, которые мы предоставили функции `forkJoin`. Каждый элемент массива, в свою очередь, будет содержать результаты отдельного Observable, от которого мы ждем результатов. Таким образом, в нашем предыдущем фрагменте кода `result[0]` будет содержать результаты `boardSizes`, а `result[1]` – результаты `boardListTypes`.

Функция `forkJoin` очень удобна, когда у нас есть несколько вызовов REST, которые необходимо завершить, прежде чем мы сможем отобразить страницу. Каждый вызов может быть добавлен в массив Observables для функции `forkJoin`, и к результатам каждого вызова можно обращаться через возвращенный массив.

Обратите внимание, что мы устанавливаем в качестве значения локального свойства `this.boardList` результат `boardListTypes` с помощью `result[1]`. Мы не используем результат `boardSizes` вообще. Однако если мы запустим нашу страницу, то увидим, что для каждой доски в нашем списке и массив `board.sizes`, и массив `board.types` были успешно заполнены.

Ключ к этой причуде в нашем коде заключается в том, что `boardSizes` и `boardListTypes` работают на основе результата исходного `getBoardsList`. Это можно увидеть, если мы сожмем наш код в соответствующие строки:

```
this.boardService.getBoardsList().subscribe(
  (result: IBoardExtended[]) => {
    let boardSizes = from(<IBoard[]>result).pipe(
      ...существующий код...
    let boardListTypes = from(<IBoard[]>result).pipe(
      ...существующий код...
```

Здесь у нас есть исходный Observable из вызова функции `boardService.getBoardsList`, доступ к которому осуществляется через аргумент `result`. Он предоставляет массив объектов `IBoardExtended` в функцию `subscribe`. Обратите внимание, однако, что и `boardSizes`, и `boardListTypes` работают с од-

ним и тем же массивом, который называется `result`, так как оба используют `from(<IBoard[]>result).pipe`. Это означает, что оба Observables фактически читают и изменяют один и тот же массив в памяти.

Модульное тестирование Observables

Прежде чем мы продолжим работу с нашим приложением, давайте кратко рассмотрим, как можно выполнить модульное тестирование `BoardListComponent` и, в частности, тестирование использования Observables. Класс `BoardListComponent` применяет `BoardService` для загрузки информации из трех разных конечных точек REST. Он вызывает функции `BoardService: getBoardList`, `getBoardSizes` и `getBoardTypes`, а также координирует эти вызовы, используя `concatMap` и `forkJoin`. Каждый из вызовов `BoardService` возвращает Observables.

К счастью, мокировать Observables в рамках модульного теста относительно просто. Давайте обновим файл `board-list.component.spec.ts`, который был автоматически создан с помощью интерфейса командной строки Angular:

```
describe('/src/app/board-list/board-list.component.spec.ts', () => {
   let component: BoardListComponent;
   let fixture: ComponentFixture<BoardListComponent>;
   let mockBoardService : BoardService;
   beforeEach(async(() => {
      TestBed.configureTestingModule({
         declarations: [ BoardListComponent ],
         providers: [BoardService, BroadcastService],
         imports: [HttpClientModule] })
      .compileComponents();
      fixture = TestBed.createComponent(BoardListComponent);
      component = fixture.componentInstance;
      mockBoardService = TestBed.get(BoardService);
   }));
```

Здесь мы определили функцию `describe`, которая будет выполнять наш набор тестов. Имя набора тестов совпадает с полным именем пути к самому файлу теста, что является рекомендуемой практикой, описанной в `главе 8` *«Разработка через тестирование»*. Далее у нас идут переменные `component` и `fixture`, как описано в `главе 9` *«Тестирование фреймворков, совместимых с TypeScript»*. Потом мы определяем переменную `mockBoardService` типа `BoardService`. Мы установим в качестве значения этой переменной экземпляр `BoardService`, который внедряет фреймворк тестирования Angular.

Затем у нас идет функция `beforeEach`, которая вызывает функцию `configureTestingModule`, содержащуюся в классе `TestBed`, и определяет три свойства: `declarations`, `providers` и `imports`. Мы уже знакомы со свойством

declarations, в котором перечислены компоненты Angular, которые необходимо определить для данного конкретного теста. В свойстве providers перечислены все классы, которые используются во фреймворке внедрения зависимости Angular. Поскольку BoardlistComponent нуждается как в BoardService, так и в BroadcastService, обе эти службы перечислены здесь. Последнее свойство, используемое в функции configureTestingModule, – это свойство imports в котором указывается, какие модули необходимо импортировать. Единственная зависимость здесь – от HttpClientModule. Как только мы вызвали функцию compileComponents, мы установили переменные fixture и component, как обычно.

Последняя строка этого фрагмента кода устанавливает в качестве значения переменной с именем mockBoardService результат вызова функции TestBed.get и предоставляет имя класса BoardService в качестве единственного аргумента. Это дает нам доступ к BoardService, которая была введена во фреймворк тестирования. Наш модульный тест выглядит так:

```
it('should load boards, sizes and types', () => {
   spyOn(mockBoardService, 'getBoardsList').and
   .returnValue( of([
      { id : 1}, {id: 2}
   ]) );
   spyOn(mockBoardService, 'getBoardSizes').and
      .returnValue( of([
         {board_id: 1, volume: 800}, {board_id: 2, volume: 600}
   ]))
   spyOn(mockBoardService, 'getBoardTypes').and
      .returnValue( of([
      {board_id: 1, board_type_id: 1},
      {board_id: 2, board_type_id : 2}
   ]))

fixture.detectChanges();

fixture.whenStable().then ( () => {
   expect(component).toBeTruthy();
   expect(component.boardList.length)
      .toBe(2, 'boardList.length should be 2');
   expect(component.boardList[0].sizes.length)
      .toBe(2, 'sizes.length should be 2');
   expect(component.boardList[0].types.length)
      .toBe(2, 'types.length should be 2');
   });
});
```

Здесь мы определили тест под названием 'should load boards, sizes and types'. Цель этого теста – убедиться, что при загрузке BoardlistComponent он

вызывает функцию `loadAndFilterBoardList`, которая, в свою очередь, вызывает функции `getBoardList`, `getBoardSizes` и `getBoardTypes`. Первая строка этого теста создает шпиона Jasmine, вызывая функцию `spyOn`, с двумя параметрами. Первый параметр – это переменная `mockBoardService`, которая является тестовым экземпляром класса `BoardService`. Второй параметр – это имя функции, за которой мы хотим следить. В данном случае это функция `getBoardsList`.

Обратите внимание на то, как мы используем этого шпиона. Мы вызываем функцию `and.returnValue`, чтобы переопределить поведение по умолчанию функции `getBoardList`. Затем мы используем функцию Observable, `of`, для создания Observable из встроенного массива, у которого есть два элемента. Вот так просто создать Observable в модульном тесте. Мы просто создали шпиона для функции, которую вызовет наш код, и возвратили Observable из встроенного массива.

Затем мы повторяем этот шаблон для вызовов `getBoardSizes` и `getBoardTypes` к `mockBoardService`.

Таким образом, использование шпионов и встроенных Observables позволяет нам контролировать точную структуру данных JSON, возвращаемых службой, и дает нам возможность создавать тестовые наборы с различными размерами массивов, различными свойствами элементов массива и даже пустыми массивами.

После того как мы настроили каждого из наших шпионов, мы вызываем функцию `detectChanges` для переменной `fixture`, которая запускает процедуры обнаружения изменений в Angular и вызывает функцию `ngOnInit`. После этого мы используем промис `whenStable`, чтобы дождаться, пока компонент завершит рендеринг, а затем можем запустить серию операторов expect.

Наш первый оператор expect просто проверяет, что сама переменная `component` была инициализирована. Таким образом мы проверяем, что класс `BoardListComponent` был успешно создан. Затем следует еще один оператор `expect`, чтобы проверить, что общее количество записей в свойстве `boardList` правильно установлено на 2. Другими словами, из функции `getBoardList` были возвращены две записи. Далее у нас идут два оператора `expect`, которые проверяют, правильно ли задано свойство первого элемента `sizes` в массиве `boardList` и правильно ли установлено свойство `types`.

В результате этого теста мы:

- создали экземпляр `BoardListComponent`;
- нашли внедренный экземпляр `BoardService` и сохранили ссылку на него;
- использовали эту ссылку для переопределения функции `getBoardsList` и возврата встроенного Observable, данные которого строго контролируются в рамках теста;
- повторили эту технику для вызовов функций `getBoardSizes` и `getBoardTypes`;

- загрузили и визуализировали компонент;
- убедились, что свойство `boardList` компонента `BoardListComponent` содержит два элемента массива;
- убедились, что свойства `sizes` и `types` этого элемента массива установлены правильно.

Наш модульный тест с использованием Observables завершен.

Крупномасштабное приложение Angular обычно имеет много различных компонентов, которые по-разному объединяются в пользовательские экраны. Каждый из этих компонентов обычно загружает данные из конечных точек API REST и может использовать различные конечные точки для загрузки всех данных, необходимых для визуализации компонента. Когда мы начинаем писать собственный код, который зависит от нескольких источников данных, как в этом примере, целесообразно написать модульные тесты, чтобы убедиться, что логика нашего приложения работает так, как ожидалось. Использование Observables внутри наших компонентов позволяет нам быстро и просто мокировать данные, которые были бы возвращены из конечной точки в рамках модульного теста.

Как мы уже видели, процесс настройки и использования Observables в среде модульного тестирования прост и интуитивно понятен. Создавая тесты, обладающие детальным контролем над данными и структурой данных, которые возвращаются службами, мы даем нашему приложению гораздо лучшую стабильность, поскольку можем тестировать крайние случаи, о которых мы, возможно, изначально и не думали.

Шаблон проектирования Domain Events

Теперь, когда у нас есть список досок на главной странице сайта, давайте посмотрим, что происходит при нажатии на определенную доску. Нам нужно будет определить, на какую доску нажали, а затем отобразить информацию об этой доске на экране справа. Этот экран будет анимирован и будет перемещаться влево, чтобы занять все пространство. Анимация и переход экранов уже готовы, поэтому нам нужно как-то получить сообщение от компонента `board-list` через компонент `rightscreen`, чтобы сообщить ему, какую доску показывать. Нам также нужно передать сообщение нашему посреднику, чтобы он перешел в правильное состояние `StateType.DetailPanel`.

Шаблон проектирования Domain Events позволяет нам генерировать события в одной части нашего приложения, а другие области нашего приложения будут реагировать на них. Когда происходит событие предметной области, части нашего приложения, которые заинтересованы в этих событиях, могут реагировать на

происходящее. Этот шаблон позволяет нам отделить области нашего приложения друг от друга и соответственно спроектировать наше решение. Шаблон проектирования Domain Events впервые обсудил *Мартин Фаулер* в своем блоге *«Событие предметной области»* (`https://martinfowler.com/eaaDev/DomainEvent.html`). Его описание этого шаблона заключается в том, что он захватывает память о чем-то интересном, что влияет на предметную область.

В современных GUI-фреймворках наш пользовательский интерфейс состоит из множества различных компонентов, каждый из которых будет отображать определенную область экрана. В нашем текущем приложении эти компоненты включают в себя компоненты `board-list`, `rightscreen`, `navbar` и `sidenav`. У нас также есть компоненты `login-panel` и `secure`. В более крупном приложении их может быть очень и очень много. Нам нужен механизм обмена сообщениями между этими компонентами. Если в компоненте `board-list` происходит интересное событие, мы должны реагировать на это событие в компоненте `rightscreen` или `secure`.

Реализация шаблона проектирования Domain Events может быть легко достигнута путем создания шины событий. По сути, это центральное место, где любой компонент может передавать событие, а любой другой компонент может эти события прослушивать. В `главе 7` *«Совместимые с TypeScript фреймворки»* мы сравнили механизмы передачи событий, доступные в каждом из обсуждаемых фреймворков, и использовали механизм Event Bus в приложении Backbone. Мы будем реализовывать аналогичную концепцию в нашем приложении на Angular.

Фреймворк Angular использует концепцию служб, которые могут быть внедрены в компонент с помощью внедрения зависимостей. Чтобы реализовать нашу шину событий, мы можем просто создать службу, которая позволяет компонентам как передавать событие, так и подписываться на него. Поэтому любой компонент, которому необходимо отправить событие или действовать в соответствии с этим событием, может просто использовать одну и ту же службу.

Давайте создадим в каталоге `services` новый файл под названием `broadcast.service.ts`:

```
import {Subject, Observable} from "rxjs";
import {filter, map} from "rxjs/operators";
import {Injectable} from "@angular/core";

export interface IBroadcastEvent {
   key: string;
   data?: any;
}

@Injectable({
   providedIn: 'root'
)
```

```
export class BroadcastService {
   private _eventBus: Subject<IBroadcastEvent>;
   constructor() {
      this._eventBus = new Subject<IBroadcastEvent>();
   }
   broadcast(key: any, data?: any) {
      this._eventBus.next({ key, data });
   }
   on(key: any): Observable<IBroadcastEvent> {
      return this._eventBus.asObservable()
         .pipe(filter(event => event.key === key
      ));
   }
}
```

Здесь мы начинаем с импорта классов `Subject` и Observable из библиотеки RxJS, а также операторов `filter` и `map`. Чтобы сделать этот класс совместимым с фреймворком внедрения зависимостей Angular, мы также импортируем декоратор `Injectable`. Затем мы определяем интерфейс с именем `IBroadcastEvent`, у которого есть свойство `key` типа `string` и необязательное свойство `data` типа `any`. Обратите внимание, что мы могли бы сильно типизировать свойство `data` как объединение типов вместо `any`, но в этом примере мы позволим свойству `data` нести данные любого типа.

Затем мы определяем службу `BroadcastService`, у которой есть единственное свойство `_eventBus` типа `Subject<IBroadcastEvent>`. Документация RxJS гласит, что `Subject` – это особый тип Observable, который позволяет значениям быть многоадресными для многих Observables. Это означает, что у нас может быть много разных Observable, являющихся объектами, которых интересуют события, и что `Subject` Observable отправит сообщение каждому из них. Observables по своей природе одноадресные. Это означает, что каждый Observable получает своего собственного Observable там, где `Subjects` являются многоадресными. Это обеспечивает идеальный механизм для шаблона Domain Events.

Наша функция-конструктор просто создает экземпляр класса `Subject`, после чего мы определяем две функции, `broadcast` и `on`.

Функция `broadcast` используется компонентом для передачи события любой заинтересованной стороне. У этой функции два параметра: `key` типа `string` и `data` типа `any`. Реализация этой функции просто вызывает функцию `next` свойства `_eventBus` с объектом, в котором находится содержимое `key` и `data`.

Функция `on` используется слушающим компонентом, чтобы зарегистрировать свой интерес к определенному событию.

У этой функции есть единственный параметр `key`, который соответствует имени интересующего ее события и возвращает Observable. Реализация функции `on`

использует функцию `filter`, чтобы возвращать только Observable, где входящая строка `key` соответствует ключу события, которое было вызвано. Это означает, что если событие было вызвано ключом '`my-event-key`', то данные получат только те слушатели, которые указали тот же '`my-event-key`'.

Оператор Observable, `filter` – суть этой реализации. Предположим, что у нас есть два слушателя, которые зарегистрировались в `BroadcastService`. Оба они подключены к Observable и ожидают доступных данных. Когда ключ соответствует и источнику событий, и слушателю, слушателю отправляются данные. Если ключ не соответствует слушателю, данные не отправляются.

Чтобы увидеть это на практике, давайте используем `BroadcastService`, дабы вызвать событие при выборе доски, а затем прослушаем это событие, чтобы показать экран с подробной информацией, связанный с той же доской.

Вызов и использование событий предметной области

Когда пользователь нажимает на конкретную доску в компоненте `board-list`, мы можем перехватить это событие, добавив обработчик кликов в файл `board-list.component.html`:

```
<div *ngFor="let board of boardList;"
   class="col-sm-4 board-detail-clickable"
   (click)="buttonClickedDetail(board)">
```

Здесь мы добавили обработчик события (`click`) для внешнего элемента `<div>`, который генерируется для каждой доски в нашем списке. Обработчик кликов называется `buttonClickedDetail`, и у него есть единственный аргумент, который устанавливается как текущий элемент массива `boardList`. Соответствующая функция в файле `board-list.component.ts` выглядит так:

```
...имеющиеся операторы import...
import {BroadcastService} from '../services/broadcast.service';

... существующий код ...
export class BoardListComponent implements OnInit {
   boardList: IBoard[] = [];

   constructor(private boardService: BoardService,
   private broadcastService: BroadcastService) { }
      buttonClickedDetail(board: IBoard) {
      this.broadcastService.broadcast(
         'board-detail-clicked', board);
}
```

Здесь мы импортировали BroadcastService как часть наших существующих операторов импорта, а также обновили функцию-конструктор, включив в нее закрытую переменную broadcastService, которая будет содержать экземпляр BroadcastService, внедряющийся в экземпляр этого класса. Затем мы определили функцию buttonClickedDetail, которая будет вызываться, когда пользователь нажимает на элемент . Эта функция просто вызывает функцию broadcast экземпляра broadcastService с двумя аргументами.

Первый аргумент соответствует свойству key, которое мы обсуждали ранее, и представляет собой простое строковое значение 'board-detail-clicked'. Второй аргумент – это класс board, который является элементом массива.

Наш компонент border-list теперь вызывает событие предметной области, чтобы указать, что пользователь щелкнул по доске и что приложение должно показать подробную информацию о ней.

Мы реализуем часть «слушателя» шаблона Domain Event в нашем компоненте secure следующим образом:

```
export class SecureComponent implements IMediatorImpl,
   AfterViewInit {
   constructor(private broadcastService: BroadcastService) {
      _.bindAll(this, [`boardDetailClicked`]);
      this.broadcastService.on(
         'board-detail-clicked')
      .subscribe(this.boardDetailClicked);
}

boardDetailClicked(value: IBroadcastEvent) {
   console.log(`SecureComponent :
      boardDetailClicked: ${JSON.stringify(value)}`);
   this.rightScreen.setBoard(value.data);
   this.mediator.moveToState(StateType.DetailPanel);
}
```

Здесь мы обновили нашу функцию-конструктор, добавив в нее закрытую переменную broadcastService, как мы видели ранее. Внутри функции-конструктора мы вызываем функцию bindAll из библиотеки Underscore, чтобы связать функцию boardDetailClicked с правильным экземпляром переменной this. Помните, что когда событие происходит на HTML-странице, контекст этого события связан с самой страницей и не связан автоматически с экземпляром класса SecureComponent, который сгенерировал страницу. Функция bindAll исправляет этот контекст таким образом, что экземпляр SecureComponent, который получает событие, сохраняется, когда событие происходит.

Внутри нашей функции-конструктора мы затем вызываем функцию on для регистрации функции, которая будет действовать как обработчик события при

возникновении события. Функции on требуется один параметр, это имя самого события, для которого в этом случае установлено значение 'board-detail-clicked'. Поскольку функция on возвращает Observable, мы должны вызвать функцию subscribe, чтобы реагировать на изменения в Observable. Вместо того чтобы определять анонимную функцию, которая будет использоваться с subscribe, мы использовали вместо этого функцию класса, которая называется boardDetailClicked.

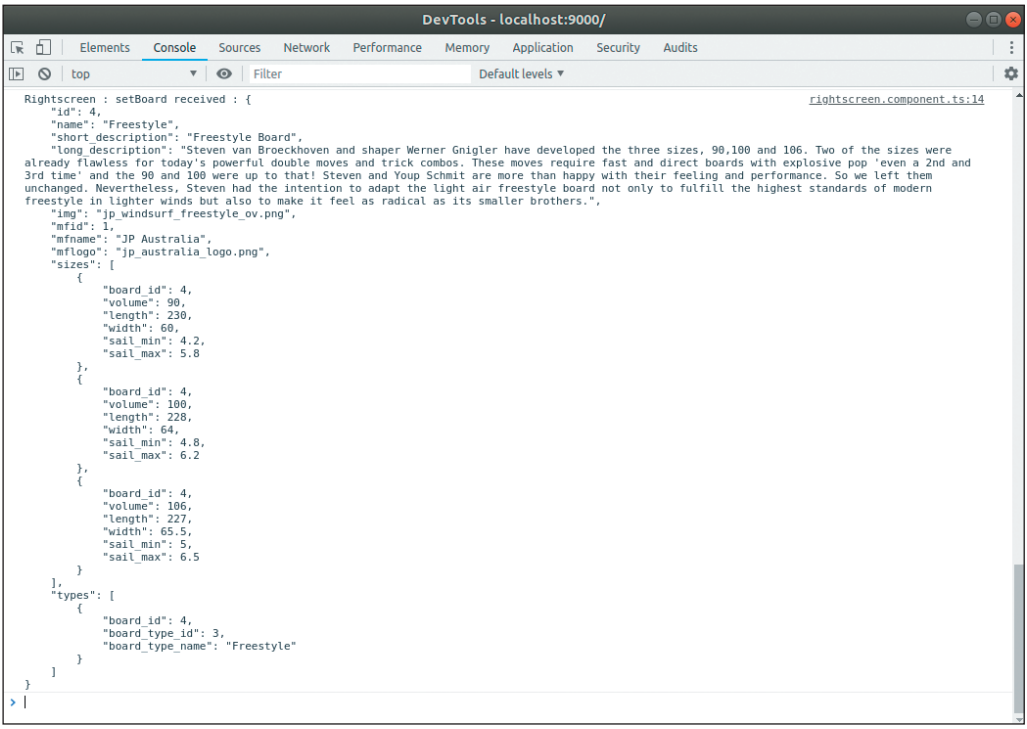
Функция boardDetailClicked получает единственный параметр с именем value типа IBroadcastEvent. В рамках этой функции мы записываем содержимое IBroadcastEvent в консоль с целью отладки, а затем вызываем новую функцию setBoard для компонента rightscreen. Помните, что компонент rightscreen – это компонент ViewChild и как таковой позволяет напрямую вызывать функции для самого класса RightscreenComponent. Функция, которую мы вызываем, называется setBoard и будет передавать свойство data полученного нами события предметной области. Фактически свойство data – это объект, который содержит все данные, относящиеся к одной доске.

После того как мы вызвали функцию setBoard компонента rightscreen, мы вызываем функцию moveToState класса Mediator для перехода нашего экрана из состояния MainPanel в состояние DetailPanel, что приведет к запуску анимации на экране и отобразит панель сведений. Нам нужно будет реализовать функцию setBoard в компоненте rightscreen следующим образом:

```
export class RightscreenComponent implements OnInit {
  board: IBoardExtended;

  setBoard(value: IBoardExtended) {

  console.log(`Rightscreen : setBoard received :
    ${JSON.stringify(value, null, 4)}`);
  this.board = value;
}
```

Здесь мы обновили класс RightscreenComponent, добавив в него локальное свойство board типа IBoardExtended. Затем мы определили функцию setBoard, которая принимает параметр value типа IBoardExtended и просто записывает сообщение в консоль, прежде чем присваивать внутреннему свойству board входящее значение.

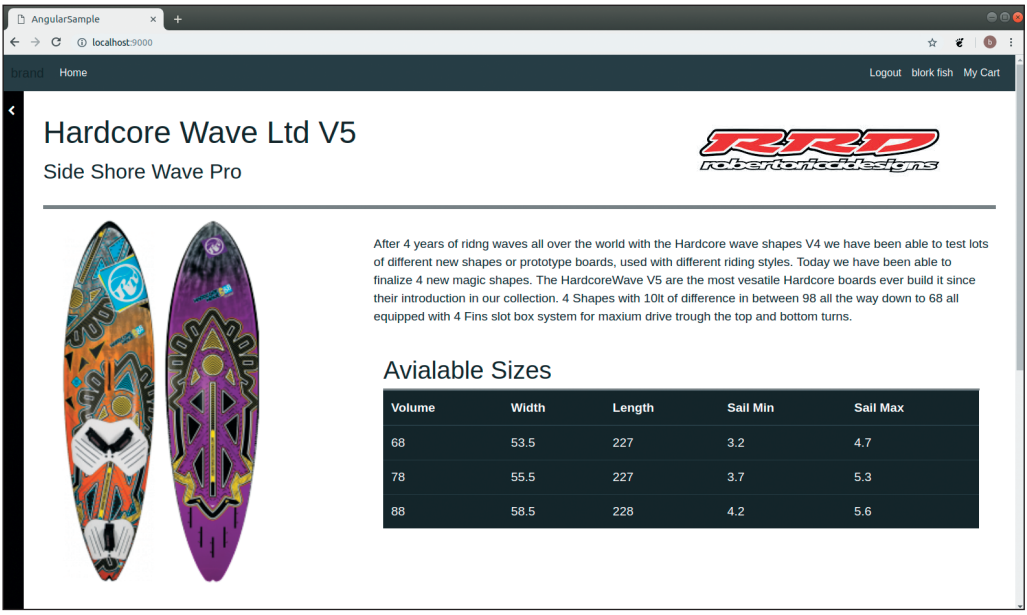
Если мы запустим сейчас наше приложение и кликнем по доске, то увидим сообщение console.log, которое будет отображаться в наших инструментах разработчика:



Здесь видно, что `RightscreenComponent` получил сообщение с использованием `BroadcastService`, которое содержит всю информацию, необходимую для отображения панели сведений о конкретной доске. Он уже включает в себя массивы `sizes` и `types`, которые были изначально загружены на экран компонента `board-list`. Имея эту информацию, мы можем обновить HTML-файл, чтобы отобразить ее для нашего пользователя.

Обратите внимание, что для краткости мы не будем обсуждать здесь полный HTML-файл, поскольку тут используются различные элементы и CSS-классы для создания эстетически приятного экрана. Пожалуйста, обратитесь к примеру кода, доступного для этой главы, чтобы просмотреть соответствующие HTML- и CSS-файлы.

Как только наши HTML- и CSS-файлы на месте, наш экран с подробной информацией выглядит так:

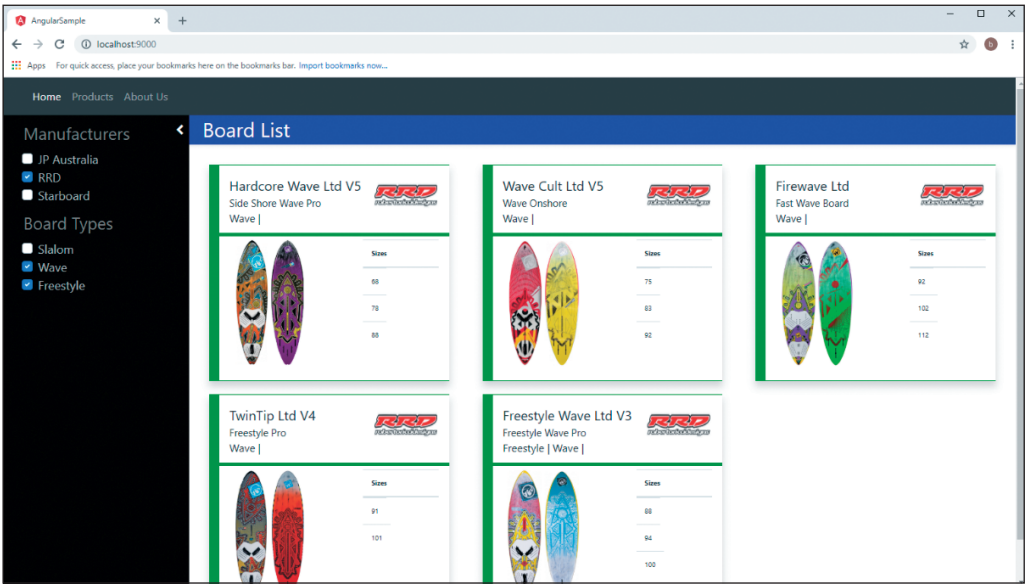


Здесь компонент `rightscreen` был обновлен, чтобы показать полную информацию о выбранной доске с дополнительной информацией о доступных размерах. Закончив с экраном со списком досок и экраном с подробной информацией, теперь мы можем взглянуть на фильтрацию списка досок из компонента `sidenav`.

Фильтрация данных

В списке досок на главной странице показаны все доступные на данный момент доски. Однако, покупая доску, виндсерферы обычно покупают доску в зависимости от того, каким видом виндсерфинга они будут заниматься. Те, кто катается по волнам, купят доску для катания по волнам, а те, кто соревнуются друг с другом, купят доску для слалома.

Давайте воспользуемся нашим компонентом `sidenav`, чтобы позволить пользователю фильтровать список отображаемых досок в зависимости от типа доски или ее производителя. У компонента `sidenav` будет две группы флажков:



Здесь видно, что пользователь выбрал для просмотра только те доски, которые изготовлены RRD и относятся к типу Wave или Freestyle.

Нам понадобятся два свойства в нашем классе `SidenavComponent` для размещения этих данных, а именно:

```
export interface IManufacturerCheck extends IManufacturer {
   checked: boolean;
}

export interface IBoardTypeCheck extends IBoardType {
   checked: boolean;
}

...определение имеющегося компонента...
export class SidenavComponent implements OnInit {
   manufacturerCheckList: IManufacturerCheck[] = [];
   boardTypeCheckList: IBoardTypeCheck[] = [];
   constructor(private boardService: BoardService,
      private broadcastService: BroadcastService) { }
```

Здесь мы добавили интерфейс `IManufacturerCheck`, который расширяет существующий интерфейс `IManufacturer` и добавляет свойство `checked` типа `boolean`. Аналогичным образом мы также создали интерфейс `IBoardTypeCheck`, который расширяет существующий интерфейс `IBoardType`. Нам нужно будет переключать свойство `checked`, когда пользователь устанавливает или убирает флажок на нашей странице. Мы также обновили класс `SidenavComponent` и вклю-

чили в него два массива ManufacturerCheckList и boardTypeCheckList для хранения этой информации.

Наша функция-конструктор была обновлена для использования фреймворка внедрения зависимости Angular для внедрения экземпляра классов BoardService и BroadcastService. Мы будем использовать класс BoardService для запроса наших конечных точек REST, а BroadcastService чуть позже для отправки события предметной области, для указания того, что пользователь изменил параметры выбора. Нам потребуется обновить функцию ngOnInit для вызова наших служб REST и загрузить эти два массива:

```
ngOnInit() {
   let manufacturerListObservable =
     this.boardService.getManufacturerList();
   let boardTypesListObservable = this.boardService.
     etBoardTypesList();

   forkJoin ([manufacturerListObservable,
     boardTypesListObservable]).subscribe((results) => {
     for (let manufacturer of <any>results[0]) {
       manufacturer.checked = true;
     }
     this.manufacturerCheckList = <any>results[0];
     for (let boardType of <any>results[1]) {
       boardType.checked = true;
     }
     this.boardTypeCheckList = <any>results[1];
   }, (err: Error) => {
     console.log(`error : ${JSON.stringify(err)}`);
   });
}
```

Здесь мы определяем две переменные Observable. Первая называется ManufacturerListObservable и является результатом вызова функции getManufacturerList BoardService. Вторая носит название boardTypes ListObservable и устанавливается как результат вызова функции getBoard TypesList.

Затем мы используем функцию forkJoin, чтобы выполнить обе эти переменные и подписаться на результаты. Как мы обсуждали ранее в этой главе, функция forkJoin будет возвращать результаты каждого Observable в массиве. Это означает, что мы можем получить доступ к результатам вызова ManufacturerListObservable, обратившись к первому возвращенному элементу массива, что видно по использованию results[0]. Аналогично мы можем получить доступ к результатам вызова boardTypesListObservable, обращаясь ко второму элементу массива, results[1].

Каждый из этих Observables будет возвращать массив, поэтому мы затем перебираем элементы каждого массива и устанавливаем для свойства `selected` значение `true`. Затем свойство `checked` привязывается к элементу `checkbox` в нашем HTML-коде:

```
<div *ngFor="let manufacturer of manufacturerCheckList;
   let i_manuf = index " class="custom-control custom-checkbox">
   <input type="checkbox" class="custom-control-input"
      id="manufacturer_{{i_manuf}}_check"
      [(ngModel)]="manufacturer.checked"
      (change)="onManufacturerChanged()">
   <label class="custom-control-label"
      for="manufacturer_{{i_manuf}}_check">
      {{manufacturer.name}}
   </label>
</div>
```

Это фрагмент HTML-кода, который используется для циклического прохождения каждого элемента массива `ManufacturerCheckList` и создания элементов `<input>` и `<label>`. Обратите внимание, что у нас есть два оператора в директиве `*ngFor`. Первый оператор – `let Manufacturer of ManufacturerList;`, который позволит нам обращаться к каждому элементу массива через локальную переменную `Manufacturer`. Второй оператор в этой директиве – `let i_manuf = index`. Этот оператор дает нам доступ к индексу в массиве для каждого элемента и делает его доступным через переменную `i_manuf`.

После этого мы создаем элемент `<input>` типа `checkbox` и присваиваем идентификатор этому элементу. Обратите внимание, что мы используем локальную переменную `i_manuf` при создании этого идентификатора. Атрибут `id` определяется как `manufacturer_{{i_manuf}}_check`, где переменная `i_manuf` будет использована в качестве параметра подстановки для генерации идентификатора, например `"manufacturer_0_check"` или `"manufacturer_1_check"`, в зависимости от значения индекса массива. Затем мы привязываем значение этого флажка с помощью директивы `[(ngModel)]` к значению свойства `selected` в массиве `manufacturerCheckList`. Обратите внимание, что мы также определили обработчик события (`change`), который связан с функцией `onManufacturerChanged`. Наш элемент `<label>` использует ту же технику подстановки индекса, чтобы связать метку с правильным элементом `<input>`.

Давайте теперь посмотрим на реализацию функции `onManufacturerChanged`:

```
onManufacturerChanged() {
   let checkedItems = _.where(this.manufacturerCheckList,
      { checked: true });

   let manufIds = _.map(
      checkedItems, (manufacturer) => { return manufacturer.id });
```

```
    this.broadcastService.broadcast('manufacturer-changed', manufIds);
  }
```

Здесь мы создаем переменную checkedItems, которая использует функцию where из библиотек Underscore, чтобы возвращать только элементы в массиве factoryCheckList, где для свойства selected установлено значение true. Помните, что каждый элемент флажка <input> на экране привязан к соответствующему элементу массива внутри массива factoryCheckList благодаря Angular, поэтому при нажатии на флажок автоматически обновится внутренняя память компонентов. Поэтому мы можем запросить внутренний массив, чтобы узнать, какие элементы в настоящее время помечены галочкой или нет.

Затем мы создаем новую локальную переменную manufIds, которая использует функцию map из библиотек Underscore для создания массива только со свойством factoryr.id и ничем больше. Это означает, что если у нашего массива есть такие значения, как 1 - JP Australia и 2 - RRD, то эта функция будет возвращать только свойства id, чтобы создать массив [1,2]. Затем мы используем boardcastService для передачи события предметной области под названием 'manufacturer-changed' с массивом идентификаторов производителей.

Обратите внимание, что реализация массива флажков для boardTypeCheckList следует той же схеме, поэтому мы не будем обсуждать ее здесь. Единственное принципиальное отличие состоит в том, что это вызовет широковещательное событие board-types-change с массивом типов досок, отмеченных галочкой. Опять же, пожалуйста, обратитесь к примеру кода, который сопровождает эту главу, чтобы получить полную версию исходника.

Теперь мы можем обратить свое внимание на фильтрацию списка досок, отображаемых на главной странице, в зависимости от того, какие флажки выбрал пользователь. Нам нужно зарегистрировать слушателя для каждого из этих событий в компоненте board-list:

```
export class BoardListComponent implements OnInit {
  boardList: IBoard[] = [];
  selectedManufacturerList: number[] = [];
  selectedBoardTypeList: number[] = [];

  constructor(private boardService: BoardService,
    private broadcastService: BroadcastService) {
    _.bindAll(this, [`onManufacturerSelectedEvent`,
      `onBoardTypeSelectedEvent`]);
    this.broadcastService.on('manufacturer-changed')
      .subscribe(this.onManufacturerSelectedEvent);
    this.broadcastService.on('board-types-changed')
      .subscribe(this.onBoardTypeSelectedEvent);
  }
```

```
onManufacturerSelectedEvent(event: IBroadcastEvent) {
  this.selectedManufacturerList = event.data;
  this.loadAndFilterBoardList();
}

onBoardTypeSelectedEvent(event: IBroadcastEvent) {
  this.selectedBoardTypeList = event.data;
  this.loadAndFilterBoardList();
}
ngOnInit() {
  this.loadAndFilterBoardList();
}
```

Здесь мы обновили класс `BoardListComponent` и ввели две новые локальные переменные `selectedManufacturerList` и `selectedBoardTypeList`. Обе эти переменные являются массивами типа `number`. Помните, что когда мы изменяем флажок для списка производителей, мы отправляем событие `manufacturer-changed`, которое включает в себя список производителей, выбранных в данный момент. Аналогично мы отправляем событие, когда список выбранных в данный момент типов досок меняется.

Эти события, однако, не зависят друг от друга. Изменение в списке выбранных производителей не влияет на текущий список выбранных типов досок, и наоборот. Это означает, что нам нужно сохранить текущее состояние обоих списков в `BoardListComponent`, чтобы иметь возможность использовать оба фильтра в списке отображаемых досок.

Наша функция-конструктор просто устанавливает обработчик событий для событий `BroadcastService`, которые будут запускать `onManufacturer-SelectedEvent` или `onBoardTypeSelectedEvent`. В этих обработчиках мы просто устанавливаем значения массива `selectedManufacturerList` либо массива `selectedBoardList` и вызываем функцию `loadAndFilterBoardList`.

Обратите внимание, что мы переместили тело кода, который был в функции `ngOnInit`, в новую функцию `loadAndFilterBoardList`, чтобы мы могли вызывать ту же самую процедуру загрузки, когда страница первоначально загружается, а также когда получаем событие предметной области.

Функция `loadAndFilterBoardList` выглядит так:

```
loadAndFilterBoardList() {
  this.boardService.getBoardsList()
    .subscribe((result: IBoardExtended[]) => {
    ...имеющиеся функции concatMap...

  forkJoin ([boardSizes, boardListTypes])
    .subscribe((result) => {
```

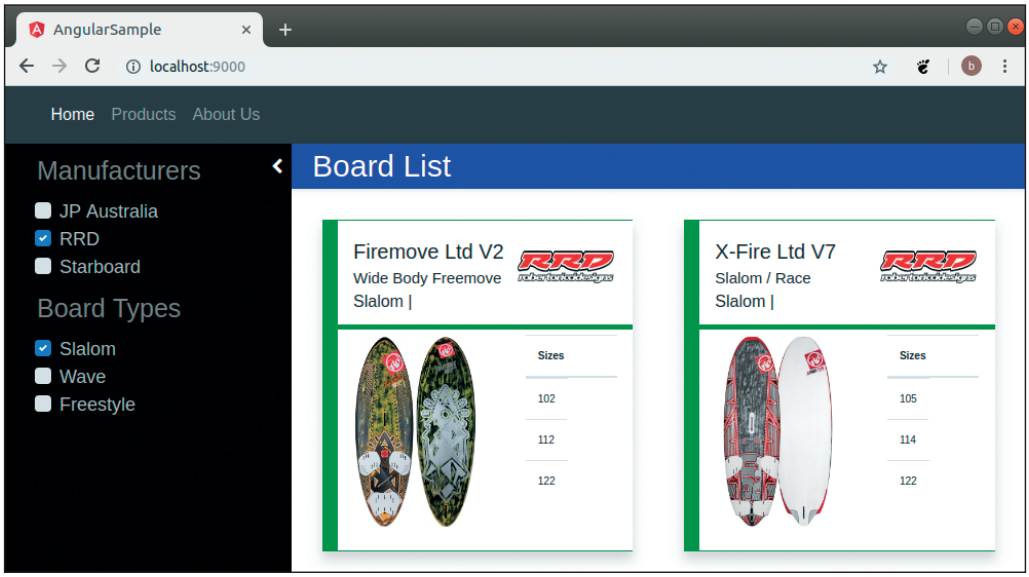
```
        // this.boardList = result[1];
        this.boardList = [];

        from(result[0]).pipe(filter(
           (board: IBoardExtended) => {
              let boardTypeIds = _.map(board.types,
              (type: IBoardType)
              => { return type.board_type_id });

              let boardTypeFound = true;
              if (this.selectedBoardTypeList.length > 0) {
                 boardTypeFound = false;
                 for (let type of boardTypeIds) {
                    if (_.contains (this.selectedBoardTypeList, type)) {
                      boardTypeFound = true;
                      break;
                    }
                 }
              }
              if (this.selectedManufacturerList.length > 0) {
                 return (_.contains
                    (this.selectedManufacturerList, board.mfid)
                    && boardTypeFound);
              } else {
                 return boardTypeFound;
              }
           })).subscribe((board: IBoard) => {
              this.boardList.push(board);
           });
        });
     });
  }
```

Здесь мы скопировали существующую реализацию `ngOnInit` в новую функцию с именем `loadAndFilterBoardList`. Как мы уже обсуждали ранее, мы подписываемся на функцию `getBoardsList` и, в рамках этого, определяем два Observables, являющихся результатом наших функций `concatMap`, которые будут загружать массивы размеров и типов для каждой доски. Затем у нас идет `forkJoin`, так что мы можем дождаться завершения обоих этих Observables, прежде чем попытаться отфильтровать результаты. Механизм фильтрации происходит внутри функции `subscribe`.

Вместо того чтобы просто присваивать все записи, полученные с сервера REST, внутренней переменной `boardList`, как мы это делали ранее, мы начинаем с очистки массива `boardList`. Затем мы используем функции `from`, `pipe` и `filter` из библиотеки RxJS для создания анонимной функции, которая будет использоваться для фильтрации результатов.

Этот механизм фильтрации делает три вещи. Во-первых, он создает массив идентификаторов типов досок из самого элемента доски и назначает его переменной `boardTypeIds`. Затем он перебирает массив выбранных типов досок и сравнивает этот список со списком типов доски, с которой он сравнивает. Если совпадение найдено, устанавливается флаг `boardTypeFound`, и цикл прерывается. Имейте в виду, что здесь мы сопоставляем связь «многие ко многим». У доски может быть много типов, например Wave и Freestyle. Если пользователь указал, что он хочет видеть все доски для катания по волнам, тогда доска с типами Wave и Freestyle все равно будет составлять пару.

После этого код фильтрации, наконец, сверяет производителя доски со списком выбранных производителей. Если совпадение найдено и тип доски совпадает, функция `filter` вернет значение `true`, и текущая доска будет добавлена в список досок, используемых для отображения. Мы можем увидеть результаты этого механизма фильтрации на приведенном ниже скриншоте:



Здесь мы выбрали только производителя, **RRD**, и тип доски, **Slalom**.

Поэтому в нашем списке досок показаны только доски для слалома, которые производятся **RRD**.

Наше приложение BoardSales теперь готово.

Резюме

Если мы оглянемся на работу, которую проделали в этой главе над приложением, то заметим, что используемая архитектура имеет качество звука. Наше приложение состоит из нескольких независимых компонентов. Каждый из этих компонентов соответствует принципу проектирования «Единственная ответственность», и они предназначены только для одной области ответственности. Мы создали компоненты отображения, которые отвечают за отображение одной области экрана приложения. Мы также использовали шаблон проектирования Domain Events для реагирования на события в нашем приложении. Один компонент может вызвать событие, но другой компонент может быть заинтересован в этом событии. Мы также используем шаблоны проектирования State и Mediator для управления анимацией и отображением различных элементов нашего экрана.

Мы обсудили использование различных методов Observables, чтобы помочь с асинхронной природой вызовов REST и ответить на ряд событий, когда нам нужно объединить данные из различных источников. Мы увидели, как использовать функцию `concatMap` для обработки потока Observables по одному элементу за раз, а также создали потоки Observables, используя функции `from` и `pipe`. Мы увидели, как ожидать завершения нескольких Observables с помощью функции `forkJoin`, как уведомлять нескольких наблюдателей с помощью `Subject`, а также как фильтровать элементы Observables с помощью функции `filter`. Мы создали модульные тесты Angular, чтобы гарантировать, что код, который мы написали для использования потоков данных Observables, имеет наивысшее качество.

Надеемся, что вам понравилось создание этого приложения и понравилось применять различные методы, которые мы обсуждали в предыдущих главах, на практике. Наконец-то мы достигли готового к применению, построенного на TypeScript одностраничного приложения промышленного уровня с использованием Angular и Node.

Указатель

A

Amazon Web Services 429, 446
AMD 3, 388, 389, 390, 397, 398, 399, 400, 402, 406, 407, 408, 410, 411, 413, 446
Angular 1 219, 232, 233, 235, 236, 237, 238, 241, 270
Apache Cordova 7
Aurelia 3, 243, 264, 265, 266, 267, 268, 269, 270, 271, 272, 273, 274, 275, 276, 277, 281, 286, 287, 291, 293, 302, 303, 309, 347, 348, 361, 362, 363, 364, 365, 367, 368, 388
Azure 30, 32, 429, 433

B

Backbone 3, 219, 225, 227, 228, 229, 230, 231, 232, 236, 237, 238, 241, 243, 249, 250, 251, 252, 253, 254, 255, 256, 257, 258, 259, 260, 263, 264, 265, 268, 269, 275, 276, 281, 286, 287, 288, 293, 301, 302, 303, 309, 347, 348, 349, 350, 351, 352, 354, 356, 361, 367, 368, 402, 404, 407, 588
Bigint 46, 80, 94, 98
Board Sales 555
Brackets 41, 507, 518, 519, 520, 528, 529, 553

C

CommonJS 3, 389, 397, 398, 399, 408, 446
concatMap 554, 576, 577, 578, 579, 580, 581, 582, 584, 600, 602

D

DefinitelyTyped 15, 222
Domain Events 554, 570, 587, 588, 589, 602

E

ECMA-262 10
ECMAScript 1, 10, 17, 67, 94, 96, 99, 116, 117, 138, 139, 143, 177, 231, 232, 242, 264, 265, 270, 389
Embedded JavaScript 191, 420
Emmet 519, 520, 553
Espruino 7
Express 3, 4, 389, 390, 396, 414, 415, 416, 417, 418, 419, 420, 421, 423, 424, 425, 426, 428, 430, 435, 440, 441, 446, 506, 507, 510, 511, 512, 513, 515, 516, 517, 525, 526, 529, 531, 533, 534, 536, 537, 538, 540, 542, 543, 551, 552, 553, 554, 559, 560, 563, 564
ExtJS 3, 219, 237, 238, 239, 240, 241

F

forkJoin 554, 581, 582, 583, 584, 596, 599, 600, 602

H

HttpClient 533, 534, 551, 553, 567

I

instanceof 99, 100, 130, 131, 132

J

Jasmine 3, 14, 233, 234, 305, 308, 309, 310, 312, 313, 314, 315, 316, 317, 318, 319, 320, 321, 322, 323, 324, 325, 327, 329, 330, 331, 332, 334, 336, 345, 346, 348, 354, 355, 356, 358, 361, 362, 364, 365, 379, 381, 387, 402, 403, 405, 406, 411, 413, 414, 440, 586
JSON Web Token 539
JWT-токен 508, 541, 546, 551, 552

M

Microsoft Visual Studio 23, 30
Mocha 308
Model-View-Controller 243
Model View Presenter 304
Model View Something 226
Model View View Model 304

N

Node 3, 4, 7, 8, 19, 20, 22, 23, 26, 30, 42, 73, 97, 177, 191, 220, 282, 308, 309, 325, 334, 335, 338, 388, 389, 390, 393, 397, 408, 414, 416, 428, 429, 430, 433, 440, 442, 444, 446, 482, 483, 484, 506, 507, 508, 511, 512, 513, 525, 557, 602
nodemailer 483, 484, 485, 487, 488, 491, 497, 498, 500
noFallthroughCasesInSwitch 210, 215
noImplicitReturns 210, 214
noUnusedLocals 210, 213
noUnusedParameters 210, 213
NuGet 219, 220, 221, 222, 223, 224, 241

O

Observable 535, 536, 537, 543, 544, 545, 546, 566, 567, 568, 577, 578, 579, 580, 583, 584, 586, 589, 590, 592, 596, 597

P

Papercut 484
POJO 243, 244, 351, 353, 377, 422, 426

S

S.F.I.A.T 58, 154
SOLID 448, 449, 450
strictBindCallApply 210, 216, 218
strictNullChecks 64, 210, 211, 212
strictPropertyInitialization 210, 212
SystemJS 3, 388, 389, 390, 408, 409, 410, 411, 412, 413, 414, 440, 446

U

undefined 63, 80, 83, 84, 85, 105, 106, 113, 116, 119, 120, 121, 122, 124, 125, 126, 127, 158, 162, 212, 213, 214, 311, 316, 360, 490, 491, 505

V

Visual Studio Code 7, 9, 19, 22, 23

W

W3Schools 455
WebStorm 9, 20, 23, 36, 37, 38, 39, 40, 41

А

абстрактные классы 99, 100, 126, 134, 447
аутентификация 345, 524, 525, 547

Б

базовые типы 46, 99, 153, 154
библиотеки JavaScript 9, 14, 15, 19, 190, 218, 222, 226, 241, 242

В

внедрение зависимости 3, 278, 281, 481, 482, 492, 497
внешняя аутентификация 547

Г

глобальные переменные 191, 218

Д

декораторы 2, 3, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 149, 150, 189, 216, 482, 498, 499, 506

З

зависимость объектов 482, 489
загрузчик модулей Require 399, 402

И

инкапсуляция 8, 15, 16, 128
интегрированная среда разработки 30, 36
интерфейсы 2, 66, 98, 99, 100, 102, 103, 104, 106, 107, 108, 112, 113, 120, 121, 126, 132, 138, 153, 154, 160, 188, 190, 200, 201, 230, 232, 236, 243, 307, 371, 390, 396, 447, 450, 490, 493, 496, 504, 568, 570

К

классы 2, 10, 16, 17, 77, 98, 99, 100,

104, 106, 108, 114, 120, 121, 122, 123, 124, 125, 126, 127, 128, 130, 132, 133, 134, 135, 137, 153, 154, 159, 161, 205, 206, 207, 208, 230, 232, 234, 237, 241, 265, 266, 278, 279, 286, 291, 370, 371, 377, 390, 392, 447, 449, 463, 465, 494, 549, 585
ключевое слово let 62, 63, 64
ключевое слово super 122, 123, 124, 127
компилятор TypeScript 3, 4, 10, 12, 13, 19, 20, 21, 41, 45, 48, 49, 50, 54, 59, 60, 64, 65, 84, 85, 103, 105, 113, 130, 150, 151, 163, 174, 192, 203, 207, 209, 213, 217, 218, 226, 240, 241, 287, 399, 493
компонент BoardList 570
контроллер 233, 243, 245, 259, 265, 271
кортеж 92, 93, 94, 166

М

модули 3, 204, 206, 282, 337, 370, 389, 396, 399, 402, 404, 405, 409, 417, 419, 426, 429, 440, 441, 446, 450, 472, 511, 585
модульные тесты 306, 307, 343, 351, 353, 361, 362, 368, 388, 587, 602

Н

наследование 2, 66, 98, 99, 100, 114, 120, 121, 122, 128, 135, 137, 158, 230, 232, 236, 237, 241, 447
необязательные свойства 101, 102, 175, 208
непрерывная интеграция 305

О

обобщения 2, 137, 138, 153, 154, 189, 216, 232, 281, 447, 450, 498, 499
обстрактные классы 125
объединенные типы 46, 80, 83, 202
одностраничное приложение 4
отправка почты 482
охранники типов 80, 81, 82, 100

П

поведенческие шаблоны 481
порождающие шаблоны 481
принцип подстановки Барбары Лисков 449, 450
принципы объектно-ориентированного программирования 396, 446, 448
промисы 138, 177, 179, 180, 181, 183, 184, 329, 438, 444
псевдонимы типов 46, 80, 82, 83

Р

разработка через тестирование 3, 233, 304, 305, 306, 584
расширенные типы 46, 79, 163

С

слабые типы 99, 102
сопоставитель 311, 314, 315
сторонние библиотеки 3, 152, 218, 219, 226, 241, 242
строгие опции компилятора 2, 64, 77, 190, 209, 218

Т

тип any 45, 56, 57, 58, 73, 78, 210, 296, 364
типизация в JavaScript 46
типы свойств 45, 66, 217

Ф

файлы определений 14, 15, 193, 219, 383
функции Lambda 429, 433, 434, 435, 436, 438, 439, 440, 441, 442, 443, 445, 446

Ш

шаблон Mediator 448, 465
шаблон проектирования Factory 2, 99, 100, 132

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «Планета Альянс» наложенным платежом, выслав открытку или письмо по почтовому адресу: **115487, г. Москва, 2-й Нагатинский пр-д, д. 6А**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.

Оптовые закупки: тел. +7 (499) 782-38-89

Электронный адрес: **books@alians-kniga.ru**.

Натан Розенталс

Изучаем TypeScript 3

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Беликов Д. А.*
Корректор *Синяева Г. И.*
Верстка *Орлов И. Ю.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16. Гарнитура «PT Serif».
Печать цифровая. Усл. печ. л. 50,7.

Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**