

ASP.NET Core

Разработка приложений

MVC, Docker, Azure, Visual Studio, C#,
JavaScript, TypeScript и Entity

Professional



Джеймс ЧАМБЕРС
Дэвид ПЭКЕТТ
Саймон ТИММС

ASP.NET Core Application Development: Building an application in four sprints (Developer Reference)

James Chambers
David Paquette
Simon Timms

Джеймс ЧАМБЕРС
Дэвид ПЭКЕТТ
Саймон ТИММС

ASP.NET Core

Разработка приложений

MVC, Docker, Azure, Visual Studio, C#,
JavaScript, TypeScript и Entity



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

2018

Джеймс Чамберс, Дэвид Пэкетт, Саймон Тиммс

ASP.NET Core. Разработка приложений

Серия «Для профессионалов»

Перевел с английского Е. Матвеев

Заведующая редакцией	Ю. Сергиенко
Ведущий редактор	Н. Римицан
Литературный редактор	Е. Самородских
Художественный редактор	С. Заматевская
Корректоры	С. Беляева, М. Молчанова
Верстка	Н. Лукьянова

ББК 32.988.02-018

УДК 004.738.5

Чамберс Джеймс, Пэкетт Дэвид, Тиммс Саймон

Ч-17 ASP.NET Core. Разработка приложений. — СПб.: Питер, 2018. — 464 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-496-03071-7

Современные разработчики занимаются построением кроссплатформенных приложений, их сопровождением и развертыванием. Чтобы облегчить им тяжелый труд, был создан новый фреймворк компании Microsoft — ASP.NET Core. Теперь в вашем распоряжении множество разнообразных библиотек с открытым кодом, более того, сам фреймворк является продуктом с открытым кодом.

Как же освоить все новые возможности, предоставляемые ASP.NET Core? Авторы объясняют решение конкретных задач на примере вымышленной компании Alpine Ski House. Каждую главу предваряет краткий рассказ о проблеме, с которой сталкивается команда разработчиков, и о том, как они эту проблему преодолевают. Вам предстоит познакомиться с архитектурой приложений, средствами развертывания и проектирования приложений для работы в облаке и многим другим.

Станьте профи в революционной технологии Microsoft — ASP.NET Core — и откройте для себя весь невероятный потенциал MVC, Docker, Azure Web Apps, Visual Studio, Visual Studio Code, C#, JavaScript, TypeScript и даже Entity Framework!

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

ISBN 978-1509304066 англ.

ISBN 978-5-496-03071-7

© 2017 by James Chambers, David Paquette & Simon Timms

© Перевод на русский язык ООО Издательство «Питер», 2018

© Издание на русском языке, оформление ООО Издательство «Питер», 2018

© Серия «Для профессионалов», 2018

Права на издание получены по соглашению с Microsoft Press. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

Изготовлено в России. Изготовитель: ООО «Питер Пресс». Место нахождения и фактический адрес:
192102, Россия, город Санкт-Петербург, улица Андреевская, дом 3, литер А, помещение 7Н. Тел.: +78127037373.

Дата изготовления: 09.2017. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —

Книги печатные профессиональные, технические и научные.

Подписано в печать 23.08.17. Формат 70х100/16. Бумага офсетная. Усл. п. л. 37,410. Тираж 1000. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».

142300, Московская область, г. Чехов, ул. Полиграфистов, 1.

Сайт: www.chpk.ru. E-mail: marketing@chpk.ru

Факс: 8(496) 726-54-10, телефон: (495) 988-63-87

Краткое содержание

Часть I. Спринт первый: Приложение Alpine Ski House 21

Глава 1. Что к чему 25

Глава 2. Факторы влияния 40

Глава 3. Модели, представления и контроллеры 52

Глава 4. Структура проекта 62

Глава 5. Сборка кода 72

Глава 6. Развертывание 88

Часть II. Спринт второй: Путешествие длиной в 1000 шагов 109

Глава 7. Сборка веб-приложений на платформе Microsoft Azure 112

Глава 8. Кроссплатформенность 130

Глава 9. Контейнеры 144

Глава 10. Entity Framework Core 160

Глава 11. Представления Razor 192

Глава 12. Конфигурация и журналирование 219

Часть III. Спринт третий: В пасти у зверя 243

Глава 13. Идентификация, безопасность и управление правами 246

Глава 14. Внедрение зависимостей 278

Глава 15. Роль JavaScript 289

Глава 16. Управление зависимостями 313

Глава 17. Стильный интерфейс 327

Глава 18. Кэширование 348

Часть IV. Спринт четвертый: Финишная прямая	359
Глава 19. Многоразовый код.....	362
Глава 20. Тестирование	377
Глава 21. Расширение фреймворка	399
Глава 22. Интернационализация	416
Глава 23. Рефакторинг и повышение качества кода.....	429
Глава 24. Организация кода	450

Оглавление

Введение	16
Для кого написана эта книга	16
Что необходимо для чтения	17
Эта книга вам не подойдет, если... ..	17
Структура книги	17
С какого места начинать чтение	18
Оформление и условные обозначения	18
Системные требования	19
Загрузки: пример проекта	19
От издательства	20
 ЧАСТЬ I. СПРИНТ ПЕРВЫЙ: ПРИЛОЖЕНИЕ ALPINE SKI HOUSE	21
Глава 1. Что к чему	25
Active Server Pages	26
ASP.NET	28
ASP.NET MVC	32
Web API	36
ASP.NET Core	38
Итоги	39
Глава 2. Факторы влияния	40
Обратная совместимость	41
Rails	42
Node.js	46
Angular и React	47
Открытый код	49
OWIN	49
Итоги	51
Глава 3. Модели, представления и контроллеры	52
М, V и С	53
Подробнее о моделях	53
Представления	55
Частичные представления	56
Контроллеры (...и действия!)	57
Не только MVC	58
Промежуточное ПО	58

Внедрение зависимостей	60
Другие усовершенствования	61
Итоги	61
Глава 4. Структура проекта	62
Лыжные склоны	64
API	67
Административное представление	68
Все вместе	69
Определение предметной области	70
Итоги	71
Глава 5. Сборка кода	72
Сборка из командной строки	73
Билд-серверы	75
Конвейер сборки	76
Сборка приложения Alpine Ski House	80
Итоги	87
Глава 6. Развертывание	88
Выбор веб-сервера	89
Kestrel	90
Обратный прокси-сервер	91
IIS	92
Nginx	94
Публикация	96
Типы сборок	98
Сборка пакета	100
Несколько слов в пользу Azure	101
Развертывание в Azure	103
Контейнерное развертывание	107
Итоги	107
 ЧАСТЬ II. СПРИНТ ВТОРОЙ: ПУТЕШЕСТВИЕ ДЛИНОЙ	
В 1000 ШАГОВ	109
Глава 7. Сборка веб-приложений на платформе Microsoft Azure	112
Что такое PaaS	113
Платформенные сервисы	113
Оснастка, уничтожение и воссоздание сервисов	116
Проектирование приложений с использованием Platform Services	117
Создание учетной записи хранения	118
Хранение изображений в BLOB-контейнерах	120
Интеграция очередей хранилища	122
Автоматизация обработки с использованием Azure WebJobs	122

Масштабирование приложений	124
Масштабирование в нескольких направлениях	125
Эластичное масштабирование	126
Другие факторы масштабирования	128
Итоги	129
Глава 8. Кроссплатформенность.....	130
Работа в Ubuntu	131
Установка .NET Core	131
Dotnet CLI.....	132
Выбор редактора кода.....	135
Проект Alpine Ski House в Linux	136
.NET Core.....	139
Итоги	142
Глава 9. Контейнеры	144
Повторяемые среды	145
Docker	149
Контейнеры Windows	154
Docker в рабочей среде.....	156
В облаке	158
Итоги	159
Глава 10. Entity Framework Core	160
Основы Entity Framework	162
Запрос на получение одной записи.....	164
Запрос на получение нескольких записей	164
Сохранение данных	165
Отслеживание изменений	165
Использование миграции для создания и обновления баз данных	166
Добавление миграций.....	167
Создание сценариев обновления для рабочих серверов	171
ApplicationDbContext	173
Расширение ApplicationUserContext	174
Контекст для карты	176
Отношения, выходящие за границы контекстов	177
Присоединение контроллера	179
Типы абонементов	185
Абонементы и проверка	187
События и обработчики событий	189
Итоги	191
Глава 11. Представления Razor	192
Создание веб-сайтов с точки зрения современного разработчика	193
Использование предыдущих достижений и находок.....	194

Роль Razor	194
Основы Razor	195
Взгляд «за кулисы»	195
Выражения в синтаксисе Razor	198
Переключение в режим кода	199
Явное использование разметки.....	200
Памятка по управлению парсером Razor.....	200
Использование других возможностей C#.....	202
Использование типов C# в представлениях.....	202
Определение модели	202
Использование данных представления	203
Работа с макетами.....	205
Основы работы с макетами.....	206
Включение секций из представлений.....	208
Определение и потребление частичных представлений.....	209
Использование расширенной функциональности Razor в представлениях.....	210
Внедрение сервисов в представления.....	210
Работа с тег-хелперами	212
Предотвращение дублирования в представлениях.....	216
Использование альтернативных механизмов представлений	217
Итоги	217
Глава 12. Конфигурация и журналирование.....	219
Расставание с web.config	220
Настройка конфигурации приложения.....	221
Использование готовых провайдеров конфигурации.....	223
Построение собственного провайдера конфигурации	224
Применение опций	227
Операции с журналом	228
Создание журналов с доступной информацией	229
Информация об исключениях	230
Журналирование как стратегия разработки	231
Уровни журналирования в ASP.NET Core	232
Применение областей действия.....	236
Структурированное журналирование.....	238
Журналирование как сервис.....	240
Итоги	242
ЧАСТЬ III. СПРИНТ ТРЕТИЙ: В ПАСТИ У ЗВЕРЯ.....	243
Глава 13. Идентификация, безопасность и управление правами	246
Эшелонированная оборона	247
Внутренние угрозы	247
Внешние угрозы	249

Скрытые данные пользователей.....	249
Аутентификация с поддержкой Azure.....	251
Идентификация в ASP.NET Core MVC.....	257
Учетные записи локальных пользователей	262
Сторонние провайдеры аутентификации	264
Включение средств безопасности с использованием атрибутов	267
Применение политик для авторизации	269
Глобальное применение политик.....	269
Определение специализированных политик	270
Специальные политики авторизации	271
Защита ресурсов.....	273
Общий доступ к ресурсам независимо от источника (CORS).....	275
Итоги	277
Глава 14. Внедрение зависимостей	278
Что такое «внедрение зависимостей»?.....	279
Ручное разрешение зависимостей.....	279
Использование контейнера сервисов для разрешения зависимостей	281
Внедрение зависимостей в ASP.NET Core.....	282
Использование встроенного контейнера	283
Использование стороннего контейнера.....	286
Итоги	288
Глава 15. Роль JavaScript.....	289
Написание хорошего кода JavaScript	290
А это вообще обязательно?.....	291
Организация кода	292
SPA или не SPA?	293
Сборка JavaScript.....	294
Bundler & Minifier	295
Grunt	296
gulp	298
WebPack	300
Какой инструмент мне лучше подойдет?.....	303
TypeScript.....	304
Компилятор ES2015 в ES5	304
Загрузка модуля.....	309
Выбор фреймворка.....	310
Итоги	311
Глава 16. Управление зависимостями	313
NuGet.....	314
Установка пакетов в NuGet	314
Инструментарий Visual Studio.....	315

npm.....	317
Добавление зависимостей	318
Использование модулей npm	318
Интеграция с Visual Studio	319
Yarn.....	321
bower	323
Добавление зависимостей	324
Обращение к ресурсам из пакетов bower	325
Итоги	325
Глава 17. Стильный интерфейс	327
Создание сайтов с таблицами стилей	328
Немного истории	329
Создание собственной таблицы стилей	331
SASS в работе со стилями	333
Основы SCSS	335
Переменные	335
Импортирование и фрагменты	336
Вложение	337
Создание примесей	339
Объединение примесей и директив	340
Рабочий процесс	341
Инструменты командной строки	341
Работа в Visual Studio Code	341
Изменение задач сборки проекта	342
Использование сторонних фреймворков	343
Расширение фреймворка CSS	343
Настройка используемых элементов фреймворка CSS	344
Применение фреймворков CSS для специализированных таблиц стилей	344
Альтернативы для фреймворков CSS	346
Итоги	347
Глава 18. Кэширование	348
Заголовки управления кэшированием	350
Использование Data-Cache	353
Кэширование памяти	354
Распределенный кэш	355
Ограничение размера кэша	357
Итоги	358
ЧАСТЬ IV. СПРИНТ ЧЕТВЕРТЫЙ: ФИНИШНАЯ ПРЯМАЯ	359
Глава 19. Многоразовый код	362
Тег-хелперы	363

Строение тег-хелперов.....	363
Тег-хелперы environment, link и script	364
Файловые шаблоны	365
Запрет кэширования.....	365
Тег-хелпер cache	367
Создание тег-хелперов	367
Работа с существующими атрибутами и содержимым	369
Компоненты представлений.....	370
Работа с компонентами представлений.....	372
Что случилось с дочерними действиями?	372
Компонент представления для связи со службой поддержки	372
Частичные представления	374
Итоги	376
Глава 20. Тестирование	377
Модульное тестирование.....	378
XUnit.....	378
Основы xUnit	379
Запуск тестов	380
Организация модульного тестирования	382
Тег-хелперы для тестирования	388
Тестирование компонентов представлений.....	391
Тестирование кода JavaScript.....	392
Другие виды тестирования.....	397
Итоги	398
Глава 21. Расширение фреймворка	399
Соглашения.....	400
Создание нестандартных соглашений.....	401
Промежуточное ПО	403
Настройка конвейера.....	403
Написание промежуточного ПО	405
Разветвление конвейера.....	407
Загрузка внешних контроллеров и представлений	408
Загрузка представлений из внешних проектов.....	409
Загрузка контроллеров из внешних сборок.....	409
Маршрутизация.....	410
Маршрутизация на основе атрибутов.....	411
Расширенная маршрутизация	412
Инструменты dotnet	412
Изоморфные приложения и сервисы JavaScript	413
Изоморфные приложения	413
Использование Node.....	414
Итоги	415

Глава 22. Интернационализация	416
Локализация текста	419
Локализация строк	419
Создание файлов ресурсов	421
Локализация представлений	422
Аннотации данных	423
Совместное использование файлов ресурсов	423
Назначение текущей культуры	424
Итоги	428
Глава 23. Рефакторинг и повышение качества кода	429
Что такое рефакторинг?	430
Оценка качества кода	432
Выбор времени для рефакторинга	434
Меры безопасности при рефакторинге	434
Разработка на основе данных	442
Пример чистки кода	443
Полезные инструменты	448
На пути к качественному коду	448
Итоги	448
Глава 24. Организация кода	450
Структура репозитория	451
Внутри исходного кода	452
Параллельная структура	452
Паттерн «Посредник»	454
Краткое введение в паттерн «Передача сообщений»	454
Реализация паттерна «Посредник»	455
Области	459
Итоги	460
Послесловие	461
Об авторах	464

Посвящаю эту книгу своей любящей жене. Спасибо тебе за поддержку! Надеюсь, мы будем проводить больше времени на природе, когда я наконец-то выберусь из этого кабинета.

Дэвид Пэкетт

Хочу посвятить эту книгу моим детям, которым приходилось играть без меня, пока я самозабвенно стучал по клавишам перед светящимся экраном, и моей жене, научившей их петь «Cat's In The Cradle» под дверью моего кабинета. Без их поддержки я был бы в лучшем случае четвертью человека. Люблю их сильнее, чем можно выразить словами.

Саймон Тиммс

Посвящаю эту книгу моей замечательной, умной, потрясающей жене. Она поддерживала меня более 20 лет, пока я гонялся за мечтами, а иногда за нашей собакой, которая сбегала посреди ночи. Посвящаю ее своим детям, которых люблю всей душой. С нетерпением жду, когда начну проводить с ними больше времени, даже если их снова придется часто отправлять в душ и они будут постоянно тащить меня на прогулку.

Джеймс Чамберс

Введение

ASP.NET Core MVC — новейший веб-фреймворк для разработчиков .NET от компании Microsoft. Это следующая версия уже знакомого фреймворка MVC, предоставляющая более широкие возможности за счет поддержки кроссплатформенной разработки и развертывания. ASP.NET Core MVC использует множество разнообразных библиотек с открытым кодом; более того, сам фреймворк построен как продукт с открытым кодом. ASP.NET Core MVC помогает разработчикам разделить бизнес-логику, маршрутизацию, сервисы и представления, а также предлагает новые методы конфигурирования и работы с расширениями. Фреймворк использует язык программирования C# и механизм представлений Razor. И будь вы опытным разработчиком .NET или новичком этой платформы, с большой долей вероятности ваши проекты будут спроектированы на основе ASP.NET Core MVC.

В этой книге рассматриваются первые спринты разработки приложения в вымышленной компании Alpine Ski House. Каждую главу предваряет краткий рассказ о проблеме, с которой сталкивается команда разработчиков, и о том, как они собираются ее преодолевать. Несмотря на присутствие элемента вымысла, в книге достаточно глубоко освещена не только функциональность ASP.NET Core MVC, но и инструментарий, используемый разработчиками для построения, сопровождения и развертывания приложений.

Кроме историй и технической информации об ASP.NET Core MVC, в книге рассматривается новая версия Entity Framework, системы управления пакетами и периферийные технологии, используемые современными веб-разработчиками. Помимо теоретического учебного материала, к книге также прилагается сопроводительный проект — тот самый, который разрабатывают программисты из Alpine Ski House.

Для кого написана эта книга

Книга проведет программиста по всем этапам создания нового приложения на базе ASP.NET Core и обеспечения доступа к нему через интернет. Многие программисты все еще далеки от мира веб-разработки или лишь незначительно со-

прикоснулись с ним посредством WebForms, несмотря на широкий спектр доступных в настоящее время программных инструментов. Книга поможет читателю овладеть навыками, необходимыми для проектирования современных приложений на активно развивающемся фреймворке. Вы изучите архитектуру приложений, средства развертывания и проектирования приложений для работы в облаке.

Что необходимо для чтения

Читатель должен уметь программировать на среднем или высоком уровне. От вас потребуется хорошее знание C#, опыт веб-разработки и понимание базовых принципов работы в Visual Studio. Опыт использования предыдущих версий MVC полезен, но не обязателен. Также пригодится умение работать в интерфейсе командной строки. После завершения книги вы сможете создавать полноценные, актуальные приложения, интегрированные с базами данных, и развертывать их в облачной инфраструктуре.

Эта книга вам не подойдет, если...

Эта книга может вам не подойти, если вы — опытный разработчик ASP.NET MVC, внимательно следивший за ходом разработки ASP.NET Core MVC или принимавший в нем непосредственное участие.

Структура книги

В книге используется необычный подход: она проводит разработчиков по отдельным спринтам в ходе разработки приложения. Рассматривается не только технология, но также процесс восстановления после ошибок и реакция разработчиков на обратную связь от пользователей. Мы начнем с пустого листа и закончим полноценным работоспособным продуктом.

Книга поделена на четыре части:

- Часть I «Спринт первый: Приложение Alpine Ski House» вводит читателя в суть ситуации: в ней описывается приложение, используемое в книге, и вымышленные персонажи истории.
- Часть II «Спринт второй: Путешествие длиной в 1000 шагов» посвящена основным функциональным аспектам приложения и настройке конвейера, с помощью которого развертывание станет происходить «на лету», а процесс будет понятным для всей команды разработки.
- Часть III «Спринт третий: В пасти у зверя», уделяет основное внимание базовой функциональности, необходимой для того, чтобы начать использовать тестовое приложение в реальных бизнес-процессах. В частности, здесь рассматривается работа с данными посредством Entity Framework Core, создание

представлений на базе Razor, настройка конфигурации и журналирование, безопасность и управление правами, и наконец, внедрение зависимостей.

- Часть IV «Спринт четвертый: Финишная прямая» описывает JavaScript и управление зависимостями, а также развивает некоторые предыдущие темы.

В конце книги рассматриваются такие важные темы, как тестирование, рефакторинг и добавление расширений.

С какого места начинать чтение

На страницах книги «ASP.NET Core: Проектируем приложение за четыре спринта (Руководство для разработчиков)» рассматривается широкий спектр технологий, связанных с фреймворком ASP.NET Core. Вполне возможно, что, в зависимости от ваших потребностей и текущего понимания стека веб-технологий Microsoft, вы захотите сосредоточиться на конкретных частях материала. Следующая таблица поможет вам решить, как лучше продолжить чтение книги.

Если вы...	Рекомендации
Новичок в разработке приложений ASP.NET Core или действующий разработчик ASP.NET Core	Обратите особое внимание на части I, II и III или читайте все главы по порядку
Программист с опытом использования предыдущих версий ASP.NET	Если вам требуется вспомнить базовые концепции — кратко пробежитесь по главам 1 и 2. Затем переходите к описанию новых технологий в оставшейся части книги
Программист, интересующийся разработкой клиентской части	Прочитайте главы 15, 16 и 17 в части IV. Просмотрите раздел, посвященный сервисам JavaScript, в главе 20
Программист, интересующийся кросс-платформенной разработкой	Весь материал книги применим в области кроссплатформенной разработки, но материал глав 8 и 9 особенно актуален

Во многих главах приводятся практические примеры, которые позволят вам опробовать только что изученные концепции. Какие бы части книги вас ни интересовали больше всего, обязательно загрузите и установите тестовые приложения на ваш компьютер.

Оформление и условные обозначения

Стиль подачи материала в книге направлен на то, чтобы информация хорошо воспринималась читателем.

- В книгу включены примеры на языке C# и в синтаксисе HTML, CSS, SCSS и Razor.
- Врезки с заголовками (например, «Примечание») предоставляют дополнительную информацию или описания альтернативных способов выполнения указанных действий.
- Знак «+» между названиями клавиш означает, что эти клавиши должны нажиматься одновременно. Например, «Нажмите Alt+Tab» означает, что вы удерживаете нажатой клавишу Alt при нажатии Tab.
- Треугольник ► между двумя и более командами меню (например, File ► Close) означает, что вы должны выбрать первую команду меню, затем следующую и т. д.

Системные требования

Для запуска приложения, прилагаемого к книге, понадобится следующее оборудование и программное обеспечение:

- .NET Core 1.0 или более новая версия для вашей платформы (доступна на сайте <https://dot.net>).
- Ваш любимый редактор кода. Мы используем Visual Studio 2015 (любое издание) или более новую версию в системе Windows и Visual Studio Code в системах Windows, Mac и Ubuntu Linux.
- SQL Server LocalDB (входит в поставку Visual Studio 2015 или более новых версий для Windows). Пользователям Linux и Mac также понадобится доступ к базе данных SQL Server, размещенной на машине с системой Windows или в Microsoft Azure.
- Компьютер с процессором от 1,6 ГГц и выше.
- Не менее 1 гигабайта оперативной памяти.
- 4 гигабайта свободного пространства на жестком диске.
- Подключение к интернету для загрузки программных продуктов и кода примера.

В зависимости от конфигурации Windows также могут понадобиться права локального администратора для установки или настройки Visual Studio 2015.

Загрузки: пример проекта

В большинстве глав этой книги приводятся фрагменты кода проекта. Весь проект доступен на GitHub:

<https://github.com/AspNetMonsters/AlpineSkiHouse>

Чтобы загрузить и запустить проект, выполните инструкции из репозитория GitHub.

ПРИМЕЧАНИЕ

Помимо кода проекта, в вашей системе должна быть установлена .NET Core 1.0 или более новая версия.

От издательства

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

Мы будем рады узнать ваше мнение!

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

ЧАСТЬ I

Спринт первый: Приложение Alpine Ski House

Глава 1. Что к чему.....	25
Глава 2. Факторы влияния	40
Глава 3. Модели, представления и контроллеры.....	52
Глава 4. Структура проекта.....	62
Глава 5. Сборка кода	72
Глава 6. Развертывание	88

Пора познакомиться с художественной частью материала, включая вымышленных персонажей, создающих приложение Alpine Ski House.

Даже самые рьяные лыжники признали: сезон подходил к концу. Это был не лучший сезон на их памяти, но он не был и худшим. В общем-то сезон был во всех отношениях неприметным. Конечно, в нем были кое-какие запоминающиеся моменты: из-за перебоев с электричеством в конце февраля пришлось пустить в ход план действий в аварийных ситуациях, который часто отрабатывался, но еще никогда не применялся. На местных каналах новостей были сообщения о детях, которые застревали в кабинках на несколько часов, но благодаря мягкой погоде им в действительности ничего не угрожало. Акции с раздачей бесплатных абонементов было достаточно, чтобы лыжники и сноубордисты приезжали снова и снова.

Весна — время, когда происходила перегруппировка постоянного персонала, а сезонные работники перебирались в теплые края. Среди постоянного персонала ходили слухи, что по крайней мере половину сезонных работников, как только

они сходили со склона, перехватывала иммиграционная служба и отправляла обратно в Австралию. Даниэль не могла представить, почему молодые сезонные работники должны были сопротивляться отправке в Австралию. Можно было смело утверждать, что в Австралии им намного интереснее, чем в сонном горном городке, который пробуждался от спячки только зимой.

Строить планы на следующий год было еще рано, и Даниэль надеялась, что у нее образуется месяц-другой свободного времени, прежде чем все завертится по новой. Она уже почти 10 лет была единственным разработчиком в Alpine Ski House, и каждый год происходило одно и то же. Большая часть ее рабочего времени уходила на поддержание работы унаследованных систем и настройку всяких мелочей, которые могли понадобиться в следующем году. Работа была не самая интересная, но в зимние месяцы в хорошие дни всем работникам удавалось провести пару часов на склоне, и это ей очень нравилось.

Даниэль открыла дверь невысокого домика, где размещалась компания Alpine Ski House, и замерла от удивления — в помещении шла какая-то бурная деятельность. Люди, которых ждали здесь не раньше чем через два часа, уже были на месте и заполняли небольшими группами все открытое пространство офиса. Озадаченная Даниэль бросила сумку, налила себе кофе и стала искать, к кому бы подойти. Похоже, весь IT-персонал собрался возле Тима — дородного руководителя IT-отдела и ее начальника. Даниэль подошла к нему.

«Даниэль! Слышала последние новости? Нас ждет интересное время, вот что я скажу», — громогласно заявил Тим.

«А что за новости?» — спросила Даниэль.

«Ты что, не в курсе? — спросил Арджун. — Наша компания только что завершила сделку по покупке курортов Thunder Valley и Mount Ballyhoo. Происходит слияние, и мы все потеряем работу!»

Два других лыжных курорта располагались всего в нескольких милях. Thunder Valley — маленькое заведение всего с тремя подъемниками, которое, тем не менее, имело своих сторонников. Этот курорт был популярен у местного населения, которое хотело отдохнуть от толп туристов в зимние месяцы. Он разительно отличался от Mount Ballyhoo — громадного курорта, охватывавшего целых три горы, с бесчисленными подъемниками и гостиницами, в которых могли бы разместиться жители двух таких городков. По выходным на склоне играли оркестры, а в толпе нередко можно было встретить таких знаменитостей, как Скотт Гатри (Scott Guthrie) и Джон Скит (John Skeet).

«Послушай, Арджун, — сказал Тим, — никто ничего не говорил об увольнении, сокращениях или о чем-то подобном. В такие времена нагрузка на IT всегда возрастает, и только лишь потому, что руководство хочет интегрировать все системы должным образом. Нужно просто подождать и понять, куда ветер дует».

Даниэль растерянно опустилась на стул. Ей совершенно не хотелось искать другую работу. Сколько вакансий для программиста найдется в таком городке с сезонным наплывом туристов? «Глупости, — сказала она себе, — пока нет смысла планировать переезд в большой город при такой неопределенности. В следующую пару недель ситуация утрясется».

Как оказалось, пару недель ждать не пришлось.

Как только закончился обед (салат из одуванчиков и грецких орехов с картофельными чипсами), Тим подошел к ее рабочему месту.

«Мы собираемся в конференц-зале. Похоже, программисты из Thunder и Ballyhoo уже там».

Допив козье молоко, Даниэль схватила ручку и блокнот и поспешила в конференц-зал. Все это было только для виду; в последние годы она не делала заметки на собраниях. Было проще лично встретиться с людьми в маленьком офисе, чем заранее планировать встречи в блокноте. Впрочем, если на горизонте маячат увольнения, лучше сразу произвести хорошее впечатление.

Большой конференц-зал использовался редко, потому что обычно просто не хватало людей, чтобы его заполнить. Но сегодня он был если и не набит до отказа, то по крайней мере не казался пустым. У одного конца стола сидели пятеро молодых парней хипстерской наружности с коктейлями-смужи самого экзотического вида.

В явном отдалении от хипстеров сидела группа людей, которых можно было безошибочно отнести к начальству. Они были загорелыми и беззаботными, и, казалось, только что вышли с поля для гольфа.

Тим подождал, пока шум стихнет, и обратился к аудитории: «У меня хорошие новости. Мы все — команда, и идти вперед нам предстоит вместе. Так что, если вы находитесь в этой комнате, значит, можно не беспокоиться; с вашей работой все в порядке. Наверняка у всех вас есть вопросы, если кто-то захочет поговорить — подойдите ко мне после собрания».

«Руководство предложило мне увеличить штат программистов после слияния, потому что нам предстоит работа над новыми интересными проектами. За несколько следующих лет мы обновим все наши программные системы, которые сейчас находятся в эксплуатации. Руководство понимает, что это серьезная работа, кое-кто из них даже читал один умный журнал и теперь знает о методологии Agile и микросервисах. Я распорядился, чтобы в будущем все экземпляры журнала сжигали до того, как они попадут в руки начальства, но сейчас нам от этого никак не уйти».

У Тима всегда были непростые отношения с новыми «великими» идеями руководства. Он работал аварийным тормозом для их безумных затей. Тим продолжил: «Первое, чего от нас хотят, — чтобы люди могли покупать билеты на подъемник

через интернет. Мне напомнили, что на дворе 2016 год и что эта функциональность должна была быть реализована давным-давно, тем более что на всех остальных курортах в мире она уже есть». Тима явно раздражали эти обобщения; вероятно, когда он получал распоряжения, дискуссия была жаркой.

«Начальство хочет увидеть прототип через месяц. Я думаю, что смогу выторговать еще неделю, если мы покажем, что работа не стоит на месте».

«Через месяц!» — подумала Даниэль. Месяц обычно требовался ей только для того, чтобы разобраться в сути проблемы. Она взглянула на хипстеров, надеясь, что они сидят с такими же кислыми лицами, как у нее, однако любители здорового питания кивали с довольным видом.

Тим, кажется, готовился наконец-то закончить свою речь. «Послушайте, ребята, нам это нужно, чтобы заработать репутацию у нового руководства. Я сделаю все, чтобы работа шла гладко. Используйте любые технологии, которые, на ваш взгляд, лучше подойдут; любые инструменты, которые вам понадобятся. Верю, что вы справитесь».

1

Что к чему

Завершив свою речь, Тим с командой гольфистов покинул комнату, и Даниэль осталась наедине с пятью программистами-хипстерами. Руководить новой группой разработки, конечно же, никого не назначили. Скорее всего, Тим тоже немного почитал злополучный журнал для IT-директоров и решил, что лучшая группа — это группа, способная к самоорганизации. Даниэль подумала, что как-нибудь стоит сунуть курьеру двадцатку, чтобы в будущем такие журналы «случайно терялись» — наверняка в следующем году она больше сэкономит на средствах от головной боли.

Хипстеры представились: Адриан, Честер, Кэндис и два Марка (Даниэль окрестила их Марк-1 и Марк-2). Все они пришли с курорта Mount Ballyhoo, руководство которого за несколько недель до этого решило отказаться от внешних разработчиков и нанять полную команду программистов в штат. Хипстеры работали вместе в стартапе в Кремниевой долине, и в Mount Ballyhoo наняли сразу всех пятерых (эта идея показалась Даниэль довольно удачной). Программисты даже не успели запустить ни один проект в Ballyhoo к тому моменту, как их выкупили. Выглядели они вполне дружелюбными и немного взволнованными.

«Мне понравилось, что руководство дает нам самим выбирать технологии, — отметил Марк-2. — Это означает, что они нам доверяют и не хотят решать за нас».

«Выбор фреймворка всегда был моей любимой частью проекта», — заметила Кэндис.

«Сейчас есть миллион отличных вариантов: Phoenix, Revel, Express, даже Meteor. А ты что предпочитаешь, Даниэль?» — спросил Адриан.

Phoenix? Express? Похоже, что авторы называли свои веб-фреймворки в честь любимых китайских ресторанов. Даниэль впервые слышала все эти названия — она была разработчиком ASP.NET до мозга костей. Когда-то она писала приложения WinForms, а весь последний год проработала с ASP.NET MVC. Отбросив сомнения, она ответила: «Обычно я используют ASP.NET».

«О, да! — воскликнул Марк-2. Даниэль показалось, что говорить на громкости ниже 90 децибел он вообще не способен. — Одна группа в Microsoft делает со-

вершенно невероятные штуки с ASP.NET Core. Ты бы видела их производительность! Полный улет!»

Даниэль не была уверена в том, что именно «полный улет» означает в этом контексте. Вроде бы ничего плохого, поэтому она кивнула в знак согласия.

«Хмм, — сказала Кэндис. — Я не ожидала, что речь пойдет об ASP.NET».

«Да ладно, это новая и крутая технология, — сказал Марк-1. — Даниэль, у тебя ведь есть опыт работы с ASP.NET. Можешь в двух словах рассказать, что к чему?»

Веб-разработка уже давно перестала быть экзотическим новшеством. На заре интернета веб-страницы были статическими, а вся навигация обеспечивалась посредством гиперссылок. Недостатки такого подхода быстро стали очевидными. Разработчику было нелегко выдержать единую тему оформления на всех страницах, а просмотр одинаковых веб-страниц вызывал у пользователей скуку.

Веб-серверы начали поддерживать технологии, благодаря которым вид веб-страницы мог изменяться в зависимости от входных данных. Одни технологии приходили и уходили (например, SSI — Server Side Includes), другие — такие, как CGI (Common Gateway Interface), — в той или иной форме продолжали существование. Название Active Server Pages, или ASP, используется уже около двадцати лет, в то время как сама технология претерпела радикальные изменения. Сейчас мы стоим перед очередным кардинальным преобразованием технологии, скрывающейся за термином ASP. Прежде чем мы погрузимся в ее изучение, давайте сначала разберемся, какой путь она прошла.

Active Server Pages

В 1996 году третья версия сервера IIS (Internet Information Services) была выпущена с поддержкой ASP первой версии. Технология ASP строится на базе технологии Active Scripting. При этом применение Active Scripting не ограничивается ASP, так как эта технология также стала частью Internet Explorer и Windows Script Host.

Технология ASP позволяет встраивать в веб-страницы сценарии, написанные на другом языке, и исполнять их. Теоретически, поддержка большинства языков обеспечивается посредством интеграции Active Scripting с технологией COM (Component Object Model). В те времена основная конкуренция в этой области шла между JScript и VBScript — двумя языками, которые пользовались хорошей поддержкой со стороны Microsoft. Впрочем, в середине 1990-х другие языки — такие, как Perl, — тоже были популярными. Язык Perl до определенной степени обеспечивал кроссплатформенную совместимость, так как он поддерживался в Linux и мог работать через CGI-шлюзы Apache. Perl был одним из самых популярных

языков для создания интерактивных веб-приложений в ранних версиях веб-сервера Apache, являясь предшественником NCSA HTTPd.

Рассмотрим пример файла, в котором в качестве языка сценариев используется VBScript. Давайте удостоверимся, что при выводе используются современные теги на базе HTML. На самом деле, ничто не мешает использовать ASP для построения современных приложений.

```
<%@ Language= "VBScript" %>

<html>
  <head>
    <title>Example 1</title>
  </head>
  <body>
    <header>
      <h1>Welcome to my Home Page</h1>
    </header>
    <section id="main">
      <%
        dim strDate
        dim strTime

        'Получение даты и времени.
        strDate = Date()
        strTime = Time()

        ' Вывод разных приветствий в зависимости от времени суток
        If "AM" = Right(strTime, 2) Then
          Response.Write "<span>Good Morning!</span>"
        Else
          Response.Write "<span>Good Afternoon!</span>"
        End If
      %>
      Today's date is <%=strDate %> and the time <%=strTime%>

    </section>
  </body>
</html>
```

Этот пример демонстрирует различные подходы к передаче данных в выходной поток. Первый подход — напрямую обратиться к ответному потоку посредством `Response.Write`, как это сделано для вывода приветствия. При втором подходе используется директива `<%= %>` — сокращенная форма записи в ответный поток. Этот способ очень похож на те механизмы, которые встречаются в других языках сценариев, например в PHP. Компилятор игнорирует все теги HTML, кроме специальных тегов, содержащих директивы. Такие теги распознаются по последовательности символов `<%`. Директива `<%@` предоставляет компилятору ASP информацию, необходимую для обработки страницы. В приведенном примере данная

директива задает основной язык для оставшейся части страницы. Простая директива `<%` обозначает простой блок программного кода.

В отличие от PHP, технология Active Server Pages позволяет организовать совместное использование библиотек посредством COM. Так страницы могут обращаться к низкоуровневым компонентам, работающим с высокой эффективностью, или пользоваться низкоуровневой функциональностью. Кроме того, эта возможность позволяет создавать библиотеки компонентов, которые улучшают структуру приложения и способствуют повторному использованию кода.

В конце 1990-х годов компания Microsoft приступила к созданию собственной платформы разработки следующего поколения. В то время она называлась Next Generation Windows Services, или NGWS. К концу 2000-х она преобразовалась в то, что мы сейчас знаем как платформу .NET. Эта платформа включала переработанную версию популярного языка Visual Basic. При переработке в язык была включена поддержка объектно-ориентированного программирования — весьма радикальное изменение. Во многих отношениях новая версия, которая стала называться VB.NET, стала совершенно новым языком.

В .NET была включена хорошо спроектированная библиотека базовых классов, которая предоставляла большой объем стандартной функциональности. На разработку BCL (Base Class Library) значительно повлияла аналогичная библиотека, заложенная в основу Java.

Также в .NET появился совершенно новый язык, который назывался C#; он также разрабатывался под сильным влиянием Java. Язык C# предоставлял C-образный синтаксис, который лучше подходил программистам с опытом работы на C и Java. Компания Microsoft интенсивно продвигала C#, и в результате этот язык стал самым популярным языком платформы .NET.

Вместе с новыми языками C# и Visual Basic .NET была представлена новая версия ASP, которая была названа ASP.NET.

ASP.NET

Уже в конце 1990-х годов стало очевидно, что Всемирная паутина — нечто большее, чем преходящее увлечение. Но, к несчастью, у компании Microsoft возникла одна проблема. Компания потратила годы на создание инструментов, позволявших проектировать приложения для настольных систем в визуальном редакторе. Разработчики привыкли придерживаться именно такого подхода в построении приложений, а компаниям совершенно не хотелось оплачивать переподготовку всего своего персонала. Взаимодействие с элементами управления обеспечивалось по модели обработки событий. Кнопке на форме назначался обработчик события, который выполнял некоторое скрытое от глаз действие, после чего в пользовательском интерфейсе (UI, User Interface) происходили изменения. Со-

общество разработчиков привыкло к этому методу разработки, и, что еще важнее, он оказался чрезвычайно эффективным. До сих пор не существует практически ни одного инструмента, способного сравниться по эффективности с принципом генерирования кода WYSIWYG (What You See Is What You Get — «Что видишь, то и получаешь»), применяемым в Visual Basic, а позднее и в WinForms. Возможно, визуальный интерфейс получался не самым впечатляющим, но если вашему продукту не требовалось ничего, кроме серых прямоугольников на сером фоне, разработка в Visual Basic 6 или WinForms давала далеко не худший результат.

Технология ASP.NET Web Forms стала попыткой перенести эту эффективность в веб-разработку. В ней была реализована аналогичная функциональность перетаскивания мышью для размещения элементов управления по сетке и взаимодействия с ними на уровне откомпилированного кода на стороне сервера.

К сожалению, модель обратной передачи (postback), которая использовалась для отправки взаимодействий на сервер, игнорировала фундаментальные различия между веб-приложениями и настольными приложениями; каждый вызов для обратной передачи данных и выполнения действия на сервере требовал полного цикла HTTP «запрос — ответ» по интернету. Несмотря на этот недостаток, технология ASP.NET Web Forms стала исключительно успешным продуктом, на базе которого работают бесчисленные сайты — как внутренние сайты компаний, так и общедоступные сайты в интернете.

Страницы Web Forms обычно состоят из двух частей: представления (view) и отделенного кода (code behind). Представление создается в специальном синтаксисе, объединяющем HTML с Visual Basic или C#. Такие файлы имеют расширение .aspx и обрабатываются компилятором ASP. По поводу того, какая информация должна содержаться в таких файлах, нет единого мнения. У разработчика имеется определенная свобода выбора, поэтому одни люди размещают в файле всю логику, а другие ограничиваются только логикой, относящейся к представлению. Известны успешные проекты, в которых использовались оба подхода. Отделенный код компилируется в DLL, это означает, что вносить изменения «на лету» не получится. Файл .aspx компилируется «на лету», а это означает, что изменения в серверную часть могут быть внесены посредством обычного текстового редактора (хотя поступать так крайне не рекомендуется).

Каждая страница реализуется в виде формы HTML. Это позволяет Web Forms легко отслеживать изменения состояния и реагировать на них обратной передачей формы. Если вы когда-либо видели сайт, который перезагружает всю страницу целиком при отведении указателя мыши от поля ввода данных, скорее всего, вы имели дело с приложением Web Forms, которое выполняло обратную передачу состояния страницы на сервер для принятия решений о том, какие элементы страницы подлежат изменению на основании введенных данных. Модель взаимодействия, мягко говоря, не блестящая, но она может быть улучшена при помощи технологий AJAX и Update Panels, которые ограничиваются обратной передачей только отдельных частей страницы.

Для сохранения единого стиля оформления на сайтах Web Forms использует эталонные страницы (master pages). Такие страницы содержат общие компоненты оформления, а разработчик может добавлять к ним разные секции в зависимости от выполняемой страницы. Такой механизм сильно экономит время разработчика, избавляя его от необходимости вносить изменения во множество мест. Приложение может иметь много разных эталонных страниц, причем поддерживается даже вложение страниц. Можно создать одну эталонную страницу для анонимных пользователей, а другую — для зарегистрированных. Вложение эталонных страниц также открывает ряд интересных возможностей: страницы могут создаваться по принципу матрешек, где каждая эталонная страница добавляет небольшую часть контента в текущую страницу (рис. 1.1).

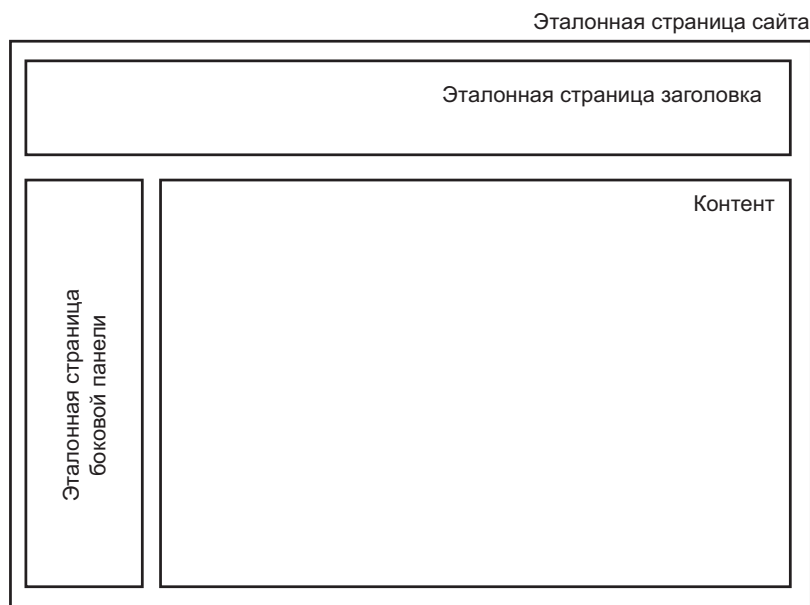


Рис. 1.1. Вложенные эталонные страницы

Элементы управления (controls) предоставляют еще один механизм повторного использования кода. Пользовательский элемент управления, обычно находящийся в файле `.ascx` и прилагаемом файле отделенного кода C# или C++, ведет себя как крошечная страница `.aspx`, которая может находиться внутри других страниц. Например, если на многих страницах вашего сайта должен присутствовать раскрывающийся список для выбора марки машины, достаточно спроектировать пользовательский элемент управления с логикой выбора. После этого вы сможете встраивать элемент везде, где он понадобится. К коду логики и отображения пользовательского элемента управления можно предоставить централизованный доступ; в этом случае вам не придется дублировать код в нескольких местах.

Проектирование эффектных элементов управления превратилось в бизнес-модель, на базе которой были основаны многие компании. Особенно популярными стали табличные элементы управления, с которыми у разработчиков всегда было много трудностей; это привело к своего рода «гонке вооружений» между разработчиками компонентов, строивших все более мощные и полнофункциональные таблицы. Что бы вам ни потребовалось, почти наверняка найдется разработчик, который предложит вам набор элементов управления, соответствующих вашим требованиям. На самом деле, если вы научитесь правильно использовать подобные сложные элементы управления, эффективность вашей работы возрастет в разы.

Несмотря на эффективность разработки, пользовательские взаимодействия в приложениях Web Forms не совсем отвечают требованиям пользователей к современным веб-приложениям. При попытке построить абстракцию, приближающую веб-программирование к настольному программированию, возникает одна проблема: интернет и настольная система — не одно и то же. Из-за множества необходимых ухищрений взаимодействие кажется громоздким и неудобным. И хотя такая модель взаимодействия может неплохо подходить для сайтов в интрасетях, вряд ли вы найдете много крупных сайтов, построенных на базе Web Forms. Программирование с Web Forms во многих отношениях напоминает попытки загнать круглую пробку в квадратное отверстие (рис. 1.2): пробка входит, но остаются зазоры.

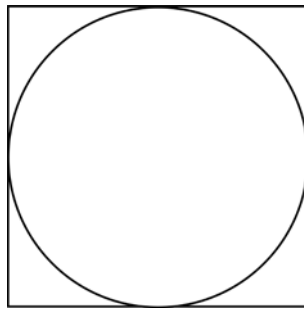


Рис. 1.2. Круглая пробка в квадратном отверстии

Одностраничные приложения (SPA, Single Page Applications) представляют собой страницы, которые находятся на стороне веб-клиента и не выполняют полного обновления в ходе навигации. К числу выдающихся примеров одностраничных приложений относятся Gmail (возможно, первое популярное SPA-приложение), Trello и Twitter. Вероятно, в процессе навигации на таких сайтах вы заметите, что перезагрузка данных ограничивается контентом, изменяющимся от страницы к странице (вместо перезагрузки всех служебных областей сайта). С точки зрения пользователя такие сайты обычно оставляют лучшие впечатления, чем традиционные сайты, потому что в ходе навигации возникает ощущение непрерывного просмотра сайта. И хотя одностраничное приложение возможно спроектировать, не используя ничего, кроме Web Forms, это будет очень трудно и займет много времени.

Состояние страницы Web Forms хранится в контейнере с именем `ViewState`. В зависимости от того, как сайт реализует `ViewState`, закодированный объект `ViewState` может присутствовать на каждой странице и передаваться от клиента к серверу и обратно при каждом запросе. Как вы можете догадаться, это существенно повышает нагрузку на канал связи, особенно если учесть, что объекты `ViewState`, передающие данные объемом в несколько мегабайт, встречаются достаточно часто. В интернете скорость передачи данных может влиять на все, от впечатлений пользователя до ранжирования результатов поиска. Проблема усугубляется повсеместным распространением мобильных устройств. Использование опции передачи данных является обыденностью для многих владельцев смартфонов и планшетов, однако слишком долгая загрузка большой страницы в сетях сотовой связи может отпугнуть пользователей.

Для преодоления некоторых недостатков Web Forms в 2007 году были введены очередные изменения — состоялся выпуск ASP.NET MVC.

ASP.NET MVC

Группы разработчиков в Microsoft всегда стараются сохранить обратную совместимость с предыдущими версиями продукта. И хотя технология ASP.NET MVC во многих отношениях превосходит Web Forms, она всегда уступала ей в приоритете и не позиционировалась как замена, оставаясь на вторых ролях в построении веб-приложений. Чтобы избежать очевидных возражений, следует сказать, что Web Forms до сих пор обладает некоторыми преимуществами перед ASP.NET MVC. Технология ASP.NET MVC работает на более низком уровне, чем Web Forms; из нее были исключены многие абстракции, которые делали Web Forms одновременно такой тяжеловесной и такой простой в использовании. Вам не обязательно выбирать какую-то одну из двух технологий, потому что MVC и Web Forms можно сочетать в одном проекте. В самом деле, есть отличный способ перевода существующего приложения с Web Forms на MVC — последовательная замена отдельных страниц.

Первая концепция Web Forms, которая была изменена в MVC, — наличие модели отделенного кода для каждой страницы. Ранее было возможно построить приложение Web Forms с четким разделением обязанностей. Однако так же просто (если не проще) строились приложения, игнорировавшие правильное разделение обязанностей. Технология ASP.NET MVC, построенная на базе паттерна MVC (Model-View-Controller, «модель-представление-контроллер»), создавала предпосылки к тому, чтобы аспекты UI изолировались в представлении, аспекты поведения — в контроллере, а аспекты хранения и передачи данных — в модели.

Встроенная система разделения обязанностей расширяет возможности тестирования и способствует повторному использованию кода. Впрочем, это всего лишь один из примеров того, как ASP.NET MVC направляет разработчиков к неизбежному успеху.

Когда ASP.NET MVC получает запрос, он почти сразу сталкивается с таблицей маршрутизации, которая передает запрос определенному приемнику — таким приемником может быть файл или фрагмент кода в контроллере. Так нарушается зависимость запросов от файлов `.aspx`, хранящихся на диске. Таблица маршрутизации открывает ряд интересных возможностей. В частности, она позволяет использовать специальные URL-адреса, которые содержат информацию о содержимом страницы для пользователей, а также для оптимизации поиска. Например, на многопользовательском (multitenant) сайте обычно создаются URL-адреса следующего вида:

`https://site.com/{Компания}/User/Add`

Сравните с более распространенной, но гораздо менее удобной версией:

`https://site.com/User/Add?companyId=15`

Также становится возможным определить несколько маршрутов, ведущих к одной конечной точке; при этом устраняется большое количество дублируемого кода.

Механизм маршрутизации обычно связывает запрос с контроллером. Контроллеры — простые объекты, происходящие от базового класса контроллера. Внутри контроллера открытые методы, возвращающие `ActionResult`, становятся потенциальными объектами для маршрутизации. Методы действий могут получать параметры обычных типов, как и методы .NET. MVC использует механизм привязки модели (model binding) для преобразования параметров из запроса HTTP в типы .NET.

Привязка модели — невероятно мощный механизм, значительно упрощающий разработку. Вы можете добавить специализированные связыватели модели для сложных объектов; впрочем, в большинстве случаев привязка по умолчанию делает все необходимое.

Метод, выбранный фреймворком, представляет собой производную от имени контроллера, метода контроллера и аннотаций методов. Например, он может иметь два метода с одинаковым именем `Add`, различающихся только в сигнатуре метода. Чтобы обеспечить правильный выбор, методы помечаются атрибутами `[HttpGet]` или `[HttpPost]`.

```
[HttpGet]
public ActionResult Add(){
    //выполнение некоторой операции
    return View();
}
[HttpPost]
public ActionResult Add(AddModel model){
    //проверка модели
    //сохранение
    //перенаправление
}
```

В часто используемой парадигме действие GET возвращает страницу формы, а действие POST получает данные этой формы, сохраняет их, а затем выполняет перенаправление на другое действие. Также существует несколько разновидностей `ActionResult`, которые могут быть возвращены действием. Вы можете вернуть результат `RedirectResult`, чтобы перенаправить пользователя на другую страницу, или результат `FileStreamResult`, чтобы отправить файл. Результат `JsonResult` выбирает тип MIME `application/json` и возвращает сериализованную версию результатов действия. С результатом `ViewResult` результат действия передается ядру представлений, отвечающему за преобразование результата в разметку HTML, которая затем возвращается конечному пользователю.

Представления («V» в аббревиатуре MVC) образуют уровень презентации и выполняют большую часть функций, которые раньше выполнялись в файле `.aspx`. Ядро представлений в MVC может заменяться, что позволяет использовать разные варианты синтаксиса. В ранних версиях MVC использовалось то же ядро представлений, что и в Web Forms, но в последних выпусках по умолчанию используется ядро представлений Razor. Razor — более современное ядро представлений, которое намного лучше интегрируется с HTML. Среди целей проектирования Razor была простота изучения и повышение эффективности работы. Безусловно, эти цели были достигнуты, и работать с Razor намного удобнее, чем со старым синтаксисом `WebView`.

Синтаксис Razor ближе к «чистой» разметке HTML, чем все предшествующие ядра представлений. В организациях, в которых веб-разработчики работают непосредственно с разметкой HTML (а сейчас эта практика становится все более распространенной), у разработчиков с переходом на Razor будет меньше проблем, чем с ядром Web Forms.

Данные, передаваемые представлением контроллеру, могут идти по двум разным путям. В первом варианте значения добавляются в `ViewBag` или `ViewData` — коллекции, доступные в представлении и контроллере. `ViewData` представляет собой словарь строк, ассоциированных с объектами, а `ViewBag` — динамический объект, к свойствам которого можно просто обращаться из программы. Разумеется, обе коллекции имеют слабую типизацию, что может потребовать преобразования типов в представлениях, а это повышает риск ошибок во время выполнения.

Второй (обычно предпочтительный) подход основан на использовании модели с сильной типизацией для передачи информации между представлением и контроллером. Модели могут проследить за тем, чтобы данные внутри этой модели отвечали ожиданиям и им были присвоены правильные имена. Ошибки времени из решения с `ViewBag` и `ViewData` заменяются ошибками компиляции, что почти всегда лучше.

Модель представляет собой простой объект CLR (POCO, Plain Old CLR Object) с коллекцией полей. MVC предоставляет в модели некоторые средства проверки данных. При помощи аннотаций из пространства имен `ComponentModel` поля мож-

но помечать как обязательные для заполнения, непустые, ограниченные определенным диапазоном и т. д. Также можно написать специализированное правило для выполнения разного рода сложных проверок: скажем, сверки с базой данных или обращения к стороннему сервису. Такие проверки могут инициироваться в процессе выполнения действия в контроллере, простой проверкой действительности `ModelState`.

Одна из претензий к ранним версиям ASP.NET MVC заключалась в том, что технология уступала Web Forms в эффективности. К сожалению, мощь пользовательских элементов управления не была легко доступной в MVC. Ситуация была исправлена с появлением редактора и шаблонов представлений. Эта великолепная возможность, которая используется реже, чем следовало бы, позволяет связать шаблон с полем модели. Встречая директиву отображения редактора для поля (как в приведенном ниже примере), Razor проверяет модель в поисках специального редактора или представления, определенного в форме вспомогательного шаблона HTML (HTML helper) или частичного представления (partial view):

```
@Html.DisplayFor(model => model.StartDate)
@Html.EditorFor(model => model.StartDate)
```

Эти директивы могут определяться по соглашениям или посредством пометки поля модели аннотацией `UHint`, определяющей загружаемый шаблон. Вспомогательные шаблоны представляют собой файлы Razor, которые могут содержать любую конкретную логику вывода или редактирования, необходимую для отображения поля. В случае с датой `StartDate` из предыдущего примера шаблон редактора может содержать элемент управления для редактирования дат. Распространяя аннотацию на несколько моделей, можно ограничить отображение и редактирование дат одним централизованным местом. Если способ отображения изменится, то изменения достаточно внести в одном файле. Это типичный пример сквозной функциональности с применением одной логики к нескольким разным классам.

Впрочем, это не единственный пример сквозной функциональности в MVC; в ней также присутствуют фильтры. Фильтр применяется посредством аннотации к контроллеру в целом или к отдельным методам внутри контроллера. Фильтры могут перехватывать и изменять любые запросы, добавлять новые или изменять существующие данные и даже вмешиваться в ход обработки запроса. Средства безопасности часто назначаются при помощи фильтровых аннотаций. Атрибуты авторизации предоставляют очень простой механизм применения сложных правил авторизации во всем приложении.

Другой пример централизации логики в приложениях MVC — промежуточное программное обеспечение (ПО) (middleware). Промежуточное ПО, построенное на базе стандарта OWIN (Open Web Interface for .NET), в основном занимает место модулей IIS (Internet Information Server), которые некогда были единственным способом перехвата всех запросов. В промежуточном ПО можно выполнять

самые разные обобщенные операции. Например, если вы захотите добавить к каждому запросу некую отладочную информацию, то для фиксации времени входа и выхода в журнале логов можно воспользоваться промежуточным ПО.

Сквозная функциональность не обязана содержаться в логике приложения — она может находиться в пользовательском интерфейсе. По аналогии с поддержкой эталонных страниц в Web Forms, MVC поддерживает макеты (layouts), которые во многом повторяют функциональность эталонных страниц. Среди новых возможностей можно отметить то, что одно представление может заполнять несколько разделов макета; например, это позволяет разместить контент и навигационную цепочку (breadcrumbs), ведущую к этому контенту, в одном файле.

MVC предоставляет больше возможностей для управления контентом и отказывается от многих несовместимых абстракций из эпохи Web Forms. Приложения, построенные с использованием MVC, обычно создают меньше проблем с тестированием, имеют более качественную архитектуру, быстрее работают с точки зрения пользователя и проще осваиваются новыми разработчиками.

Web API

Почти мгновенный рост популярности ASP.NET MVC вдохновил группу ASP.NET на создание новой технологии: Web API. Если ASP.NET MVC пытается воспользоваться популярностью Web Forms, Web API пытается проделать то же самое с сервисами SOAP (Simple Object Access Protocol), созданными в виде файлов .asmx. Расширение .asmx использовалось первым поколением технологий удаленного вызова методов через HTTP, предложенных Microsoft. Парадигма веб-сервисов SOAP обладала широкими возможностями, но очень быстро стала чрезмерно усложненной. Такие стандарты, как WS-Security и WS-BPEL, оказались слишком сложными и невразумительными. Для поддержки этих стандартов компания Microsoft создала WCF (Windows Communication Foundation) — исполнительную среду и набор API, которые довели сложные стандарты WS-* до такой степени сложности, что ни один нормальный человек не мог в них разобраться.

Возникла необходимость в более простом механизме обмена данными между сервисами, а также веб-клиентами и серверами. К счастью, в то время росла популярность концепции передачи репрезентативного состояния (REST, Representational State Transfer) из диссертации Роя Филдинга (Roy Fielding). REST — простой механизм передачи информации о действиях с множеством объектов, основанный на использовании HTTP-команд. В веб-сервисе SOAP создается конечная точка с именем вида `AddUser`, которая получает параметры, необходимые для создания нового пользователя, упакованные в конверт (envelope) SOAP. Конверт содержит тело сообщения и, возможно, некоторые заголовки. Далее сервер возвращает ответ, подтверждающий, что добавление пользователя прошло успешно. Запрос может выглядеть примерно так:


```

POST /User HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 299
SOAPAction: "http://www.w3.org/2003/05/soap-envelope"

<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  <soap:Header>
</soap:Header>
  <soap:Body>
    <m:AddUser xmlns:m="http://www.example.org/user">
      <m:FirstName>Glenn</m:FirstName >
      <m:LastName>Block</m:LastName >
    </m: AddUser >
  </soap:Body>
</soap:Envelope>

```

Как видите, сообщение довольно сложное. Если сервер сможет правильно обработать запрос, скорее всего, он вернет код результата HTTP 200. REST-сообщение для той же цели может выглядеть так:

```

POST /User HTTP/1.1
Host: www.example.org
Content-Type: application/json; charset=utf-8
Content-Length: 75

{ "FirstName": "Glenn", "LastName": "Block" }

```

Большой интерес, нежели четко выделенная информационная нагрузка, может представлять другой момент: по всей вероятности, сервер ответит кодом состояния 201. Если этот код вам не знаком, он означает «создано успешно». REST старается отсечь как можно большую часть шума SOAP, используя соглашения из спецификации HTTP. Например, для удаления элемента используется команда DELETE, а для получения информации — команда GET.

WCF позволяет создавать REST-совместимые веб-сервисы посредством создания проектов WCF Web API. Рациональные значения по умолчанию были заданы для многих параметров еще в ранней, чрезвычайно упрощенной версии WCF, а теперь доступны в той WCF, которую мы знаем сейчас. Проект WCF Web API со временем преобразовался в проект Web API (без WCF), который также стал более простым.

При первом взгляде на код проекта Web API разработчик может принять его за приложение MVC, и такое заблуждение вполне простительно. В проекте Web API присутствует папка **Controllers**, а также папка **Models** — как и в полноценном приложении MVC. Папки **View**, а также некоторых других (таких, как **Content** и **Scripts**) нет, но в них должны храниться ресурсы, используемые для прорисовки пользовательского интерфейса; в проекте Web API они не существуют.

Обратившись к контроллеру в проекте Web API, можно заметить, что имена действий соответствуют HTTP-командам, на которые они реагируют. Например, метод `Post` будет выполняться по событию получения контроллером запроса HTTP POST. По умолчанию генерируется метод `Post`, который выглядит так:

```
public async Task<IHttpActionResult> Post(User user)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    // TODO: здесь добавляется логика создания.

    // return Created(user);
    return StatusCode(HttpStatusCode.NotImplemented);
}
```

Как видите, имя метода соответствует команде, на которую этот метод реагирует.

WCF поддерживает самостоятельный хостинг (self-hosting), то есть возможность выполнения вне IIS; это наследие перешло в Web API. Это означает, что вы можете легко и быстро создать приложение Web API, которое можно развернуть на двоичном уровне.

Пожалуй, самая интересная особенность Web API заключается в том, что это один из первых проектов с открытым исходным кодом от компании Microsoft. Исходный код был полностью опубликован и в настоящее время распространяется на условиях лицензии Apache 2.0. Проект является частью .NET Foundation — некоммерческой независимой организации, которая ставит своей целью содействие открытой разработке в экосистеме .NET.

Большая часть функциональности Web API воспроизводится в базовом варианте ASP.NET MVC. Существует некоторая путаница относительно того, когда следует использовать контроллер Web API, а когда лучше использовать контроллер MVC; ведь при определенной настройке контроллер MVC работает практически так же, как контроллер Web API в простых ситуациях. Web API и MVC могут одновременно использоваться в одном проекте.

ASP.NET Core

Эволюция ASP.NET продолжается, и следующий ее шаг — ASP.NET Core. Возможно, вы также слышали названия ASP.NET vNext, ASP.NET 5 или ASP.NET MVC6; все они использовались на ранней стадии. Разработка следующей версии ASP.NET и ASP.NET MVC велась чрезвычайно открыто. Хотя предыдущие версии ASP.NET MVC распространялись с открытым кодом, сама разработка все еще оставалась закрытой. В проекты можно было передавать свой код, но процесс был

достаточно сложным. Можно даже сказать, что внешнее участие не поощрялось. А это означало, что для включения функции, очень важной для вашего проекта, вы все еще зависели от милости Microsoft.

Разработка ASP.NET Core велась на GitHub — ресурсе, где каждый желающий может обращаться с запросами на добавление возможностей или исправление ошибок. Сообщество независимых разработчиков внесло в проект значительные изменения, и хотя руководство проекта по-прежнему осуществляется компанией Microsoft, этот продукт получился намного более качественным, чем если бы он разрабатывался внутри корпорации.

Если вы захотите внести свой вклад в ASP.NET, откройте страницу <https://github.com/aspnet/home> и прочитайте, как именно это можно сделать.

ASP.NET Core принципиально отличается от всех предыдущих версий ASP.NET. Это масштабная инициатива по модернизации, которая должна обеспечить высокую конкурентоспособность ASP.NET на фоне мощных современных веб-фреймворков нашего времени — таких, как разработки сообществ Node, Elixir и Go.

ASP.NET Core идет рука об руку с инициативой по модернизации исполнительной среды .NET, которая называется .NET Core. Эта инициатива ориентирована на целый ряд усовершенствований, важнейшее из которых, пожалуй, состоит в том, что .NET Core поддерживается не только в Windows, но и в других операционных системах. Таким образом, вы можете разработать свое приложение в MacOS и развернуть его через образ Docker в группе машин Linux, работающих в облаке Azure.

ASP.NET Core — выдающийся технический проект, который, пожалуй, заслуживает как минимум отдельной книги.

Итоги

Стек веб-технологий Microsoft включает целый ряд разных технологий на стороне сервера. Правильный выбор технологии в значительной степени зависит от требований проекта. В одних ситуациях Web Forms до сих пор может оставаться правильным вариантом; другие ситуации требуют более широких возможностей настройки вывода, и для них лучше подходят MVC и Web API.

По мере развития стека технологий Microsoft другие технологии веб-разработки тоже не стояли на месте. Какой бы большой и влиятельной ни была компания Microsoft, она не может единолично диктовать будущее веб-технологий. Следовательно, очень важно понимать, какое место отведено технологии ASP.NET в общей картине веб-разработки и какие факторы повлияли на ее развитие. Эту тему мы рассмотрим в следующей главе.

2

Факторы влияния

Даниэль на мгновение прикрыла глаза; казалось, она уже целую вечность говорит об ASP.NET. Двадцать лет истории ASP.NET свелись к часовому рассказу и трем маркерным доскам, изрисованным стрелками и корявыми фигурами.

До этого Даниэль никогда не задумывалась об истории ASP.NET в целом. Было интересно проанализировать прогресс этой технологии.

«Право, не знаю, — задумчиво произнесла Кэндис. — Все это выглядит так, что мы завязнем в стеке технологий, которые поддерживает только Microsoft. И боюсь, что нам придется работать не так, как считаем правильным мы, а так, как Microsoft того хочет».

Даниэль знала, что не все хотели того же, чего хочет Microsoft, — но в их случае дело обстояло совсем иначе, верно? Есть веб-серверы, есть веб-страницы, так? Технология Web Forms была немного специфичной, зато MVC очень походила на все остальное. Но прежде чем она успела озвучить свои мысли, ее, как обычно, опередил Марк-1.

«Ну конечно, десять лет назад все было именно так. В Microsoft тогда кое-что делалось иначе. Помните ту штуку в IE4 — где можно было использовать переходы между страницами в стиле PowerPoint¹?»

«Но эта новейшая версия ASP.NET Core на самом деле берет множество лучших идей из других фреймворков и делает их своими. Шаблон по умолчанию уже идет с поддержкой gulp, npm и bower».

«Правда?» — спросила Даниэль. Она не обращала внимания на такие нюансы нового фреймворка.

«Ага! — сказал Марк-1, который быстро превращался в сторонника ASP.NET Core. — Может, ранние версии ASP.NET и создавались независимо, но на формирование ASP.NET Core повлияли многие другие фреймворки, например Rails».

¹ [https://msdn.microsoft.com/en-us/library/ms532847\(v=vs.85\).aspx#Interpage_Transition](https://msdn.microsoft.com/en-us/library/ms532847(v=vs.85).aspx#Interpage_Transition).

Остров Мадагаскар расположен достаточно далеко от африканского континента, что затрудняет перемещение растений и животных между двумя массивами суши. В результате на Мадагаскаре встречаются интереснейшие и совершенно уникальные представители земной флоры и фауны. Многие места лишены преимуществ подобной изоляции, в результате чего их растительные и животные виды находятся под сильным влиянием «гостей», приходящих за сотни и даже тысячи миль. Технология ASP.NET давно жила на одном из таких изолированных островов. ASP.NET MVC была более открытой версией, заимствующей ряд идей из других популярных фреймворков, а ASP.NET Core окончательно порывает с изоляцией и подвергается сильному влиянию других веб-фреймворков и технологий. В этой главе рассматриваются некоторые причины, по которым группа разработки ASP.NET приняла те или иные решения, а также указывается, из каких других проектов эти идеи были позаимствованы.

Обратная совместимость

Говорят, что ASP.NET Core в значительной степени отходит от тех принципов, по которым ASP.NET работала прежде. Те же претензии предъявлялись к технологии ASP.NET MVC, когда она была выпущена впервые, и, скорее всего, к технологии ASP.NET Web Forms, когда она вошла в мир, в котором доминировала простая технология Active Server Pages. Несомненно, каждый из таких переходов означал расставание с предыдущим поколением инструментальной поддержки. С другой стороны, каждое изменение имело добавочную природу и строилось на базе предыдущего поколения. Например, объект `HttpRequest` достаточно стабильно существует с начала 2000-х годов.

Microsoft Windows — выдающийся пример стремления к сохранению обратной совместимости. Приложение, написанное для Windows 95, с большой вероятностью будет работать без каких-либо изменений в новейшей версии Windows. Конечно, такая обратная совместимость не дается даром. Каждое изменение, вносимое в Windows API, приходится тщательно проверять — не нарушит ли оно работоспособность существующих приложений? Как нетрудно представить, это означает гигантский объем тестирования даже для небольших изменений. Кроме того, балласт требований к поддержке унаследованных приложений препятствует эволюции операционной системы.

Команда разработки ASP.NET приложила массу усилий к тому, чтобы в процессе использования разных продуктов возникало ощущение присутствия знакомых имен и концепций. Она постаралась, чтобы переход на новый продукт был простым и понятным для разработчиков. На ASP.NET Core прежде всего влияли предыдущие версии ASP.NET, а программисту, работавшему с предыдущей версией ASP.NET MVC, технология ASP.NET Core MVC наверняка покажется знакомой. Все фундаментальные типы — `Model`, `View`, `Controller` — всё так же используются в ней. Razor по-прежнему остается движком представлений по умолчанию, а дей-

ствия контроллеров по-прежнему возвращают типы с теми же именами. Все вносимые изменения были тщательно продуманы. Среди изменений ASP.NET Core MVC можно отметить применение передовых решений в разработке на JavaScript, а также улучшенную поддержку тестирования и полноценную поддержку внедрения зависимостей.

Рассмотрим другие технологии, повлиявшие на формирование фреймворка.

Rails

Ни одно обсуждение веб-фреймворков на базе модели MVC (Model-View-Controller) не обходится без упоминания Ruby on Rails. Паттерн MVC появился достаточно давно, еще в 1970-е годы, и его применение в области веб-технологий было неизбежным. Ruby on Rails реализует MVC с использованием языка программирования Ruby. Фреймворк Ruby on Rails (далее просто Rails) был разработан компанией 37 Signals в начале 2000-х для упрощения разработки программы управления проектами Basecamp, и в дальнейшем создатели перешли на открытое распространение данного продукта.

На момент выпуска Rails понятие веб-разработки представлялось чем-то неопределенным. Несмотря на то что технология ASP.NET существовала, «официально» работать с ней можно было только через Web Forms. В среде разработчиков Java появился фреймворк Spring — мощный инструмент, включавший ряд инструментальных средств для MVC. Поддержка MVC в Spring в основном появилась как реакция на Jakarta Struts — фреймворк, построенный на базе сервлетов, которые тогда считались официальным подходом Sun к веб-программированию на стороне сервера. Конечно, язык PHP был так же популярен в то время, как и сегодня. Однако ранние версии PHP были весьма несовершенны. Фреймворки того времени были немногочисленны и спроектированы в основном с сильной взаимосвязью компонентов внутренней среды, что означало неминуемые сложности с тестированием и расширением.

Rails вдохнул струю свежего воздуха в эту область разработки. Фреймворк задавал жесткие рамки ограничений: фактически разработчик мог действовать только по правилам Rails, и отклониться от этого пути было очень трудно. Вместо поддержки сложных параметров конфигурации и бесконечной возможности настройки Rails отдает предпочтение использованию соглашений. Это свойство охватывает все аспекты Rails. Например, структура каталогов приложения Rails, показанная на рис. 2.1, генерируется инструментарием Rails и остается неизменной практически во всех проектах Rails.

Контроллеры находятся в каталоге `/app/controllers`, их именуют, добавляя `_controller` к названию. Представления находятся в каталоге `/app/views`. При добавлении новой сущности следует использовать команду Rails, чтобы сгенерировать не только контроллер, но и представления, вспомогательные классы и даже модульные тесты.

















 app	2016-05-16 10:25 PM	File folder	
 bin	2016-05-16 10:25 PM	File folder	
 config	2016-05-16 10:25 PM	File folder	
 db	2016-05-16 10:25 PM	File folder	
 lib	2016-05-16 10:25 PM	File folder	
 log	2016-05-16 10:40 PM	File folder	
 public	2016-05-16 10:25 PM	File folder	
 test	2016-05-16 10:25 PM	File folder	
 tmp	2016-05-16 10:25 PM	File folder	
 vendor	2016-05-16 10:25 PM	File folder	
 .gitignore	2016-05-16 10:25 PM	Visual Studio Code	1 KB
 config.ru	2016-05-16 10:25 PM	RU File	1 KB
 Gemfile	2016-05-16 10:25 PM	File	2 KB
 Gemfile.lock	2016-05-16 10:25 PM	LOCK File	3 KB
 Rakefile	2016-05-16 10:25 PM	File	1 KB
 README.rdoc	2016-05-16 10:25 PM	RDOC File	1 KB

Рис. 2.1. Пример структуры каталогов, созданной инструментарием Rails

```
rails g controller menu
```

```
Running via Spring preloader in process 15584
```

```

create  app/controllers/menu_controller.rb

invoke  erb

create  app/views/menu

invoke  test_unit

create  test/controllers/menu_controller_test.rb

invoke  helper

create  app/helpers/menu_helper.rb

invoke  test_unit

create  test/helpers/menu_helper_test.rb

invoke  assets

invoke  coffee

create  app/assets/javascripts/menu.js.coffee

invoke  scss

create  app/assets/stylesheets/menu.css.scss
```

Метод определения структуры на основе соглашений, вероятно, покажется разработчикам знакомым, потому что он также использовался группой разработки ASP.NET при создании ASP.NET MVC. В мире Web Forms не существовало общей структуры каталогов и не было никаких соглашений об именах. В результате каждое приложение Web Forms было уникальным. Структура проекта зависела исключительно от фантазии разработчиков, занимавшихся им. Слишком часто неопытные разработчики просто добавляли файлы `.aspx` в корневой каталог, в результате чего появлялись каталоги с сотнями файлов и без какой-либо организационной структуры. Новым разработчикам было очень трудно подключиться к проекту и быстро разобраться в происходящем.

В ASP.NET MVC шаблон нового проекта имеет определенную структуру каталогов. Даже в контроллерах влияние соглашений на подход к конфигурированию очевидно. Чтобы вернуть объект в результате действия, обычно используется следующая запись:

```
public ActionResult Index()
{
    return View();
}
```

Даже в этом коротком фрагменте задействованы многочисленные соглашения. Во-первых, любой публичный (`public`) метод, который возвращает `ActionResult`, считается открытой конечной точкой. Во-вторых, метод называется `Index`, а это специальное имя связано с корневым маршрутом вмещающего контроллера. Далее, метод `View` не получает аргументов. По соглашению файлом `cshtml`, возвращаемым этим действием, будет файл с именем `Index.cshtml` из подкаталога `/Views`, соответствующего имени контроллера.

Если бы команда разработки ASP.NET вместо этого решила, что все эти объекты должны быть заданы явно, тот же простой метод разросся бы до следующего вида:

```
[ActionMethod]
[Route("/Menu")]
public ActionResult Index()
{
    return View("~/Views/Menu/Index.cshtml");
}
```

Код стал куда более объемистым. Недостатком использования соглашений вместо явной конфигурации является высокий порог вхождения для новых разработчиков. Впрочем, этот порог для ASP.NET MVC достаточно преодолеть всего один раз — вместо того, чтобы изучать новую структуру для каждого проекта, с которым доведется соприкоснуться разработчику.

И в Rails, и в ASP.NET MVC соглашения можно переопределить при необходимости. Переопределение следует применять осмотрительно и только по веским причинам, потому что нестандартное использование технологии, отдающей предпо-

чтение соглашения перед конфигурацией, наверняка собьет с толку неопытных разработчиков.

Другая идея, унаследованная от Rails, — маршрутизация. В эпоху Web Forms веб-сервер искал на диске файлы, имена которых соответствовали имени файла в запросе HTTP. Существовала возможность перехвата запросов и перенаправления их на другие страницы для обработки тех ситуаций, когда исходные файлы оказывались перемещены в другое месторасположение. Впрочем, правила перепределения не были нормой; использование модуля перепределения в проекте встречалось довольно редко.

В ASP.NET MVC появился механизм маршрутизации, с большой долей вероятности построенный по образцу Rails. Перед началом создания приложения устанавливается набор правил, которые могут связывать запросы с файлами на диске или действиями внутри контроллера. Не все действия в приложении требуют явного отображения: таблица маршрутизации — еще один аспект, в котором действует принцип «соглашения важнее конфигурации». Возьмем следующий фрагмент:

```
routes.MapRoute("Default", "{companyName}/{projectName}/{controller}/{action}/{id}",  
                new { id = UrlParameter.Optional } );
```

Фрагмент создает в приложении маршрут, который связывает обобщенный набор URL-адресов с контроллерами и действиями. Этот конкретный маршрут также извлекает значения переменных с названиями компании и проекта. Впоследствии эти переменные можно использовать прямо в действии, вместо того чтобы извлекать их из строки запроса.

Помимо некоторых удобных соглашений, маршрутизация позволяет использовать «наглядные» URL-адреса. Например, вы можете использовать запись:

/AlpineSkiHouse/Rebrand/user/details/12

Сравните с такой записью:

/user/details?id=12&companyName=AlpineSkiHouse&projectName=Rebrand

«Наглядные» URL-адреса не только удобнее воспринимаются пользователями, но и лучше подходят для поисковой оптимизации.

Другой принцип проектирования, лежащий в основе Rails, — нежелательность повторения. Примером служит использование модели на всем пути от пользовательского интерфейса и контроллера до модуля преобразования данных (database mapper) ActiveRecord. Та же идея встречается в ASP.NET MVC в сочетании с Entity Framework.

Фреймворк Rails был и остается популярным, поэтому вполне логично, что группа разработки ASP.NET позаимствовала некоторые удачные идеи. Эти идеи проникли как в ASP.NET MVC, так и во многие другие фреймворки и инструменты.

Отношения между такими технологиями, как Rails и CoffeeScript, были в значительной мере симбиотическими; развитие одной направляло развитие другой. То же самое можно сказать и об отношениях между ASP.NET MVC и Rails.

Node.js

Популярный блоггер Джефф Этвуд (Jeff Atwood) сформулировал принцип, который он назвал «законом Этвуда»: «Всякое приложение, которое может быть написано на языке JavaScript, будет со временем написано на языке JavaScript». Этот довольно комический закон снова и снова доказывал свою истинность, несмотря на то что он появился за несколько лет до появления Node.js. Node — мультиплатформенная, управляемая событиями, однопоточная исполнительная среда для JavaScript. Среда Node поддерживает практически любую мыслимую функциональность и стала исключительно популярной при построении веб-приложений. И если технология Rails дала толчок для разработки ASP.NET MVC, Node играет аналогичную роль в разработке ASP.NET Core 1.0. Симбиотические отношения между Node и ASP.NET Core MVC 1.0 еще более сильные, потому что многие инструменты, использованные при построении ASP.NET Core, используют исполнительную среду Node.

Еще до разработки ASP.NET Core влияние среды Node можно было заметить в реализации веб-сокетов ASP.NET — библиотеке SignalR. Node популяризирует идею отправки данных по инициативе сервера и использования WebSockets для организации обмена данными между веб-браузерами и веб-серверами в реальном времени с использованием пакета `Socket.io`. Пакет `Socket.io` был достаточно революционным и оказал большое влияние на разработку SignalR. SignalR изначально представлял собой разработку с открытым исходным кодом, реализующую ряд технологий, таких как длинный опрос (long polling), бесконечный фрейм (forever frame) и WebSockets. В конечном итоге библиотека SignalR стала официальной реализацией Microsoft. И хотя SignalR не входит в первый выпуск ASP.NET Core, она занимает важное место в последующих выпусках.

На базе Node.js построено огромное количество инструментов. Такие инструменты веб-разработки, как Grunt, Gulp и даже TypeScript, работают на базе Node.js. Даже редактор с открытым кодом Visual Studio Code базируется на технологии Electron, которая в свою очередь базируется на Node.js. Гибкость Node позволяет создавать минималистичные реализации. Простой веб-сервер может быть построен всего за несколько строк кода:

```
var http = require('http');
http.createServer(function(req, res){
  ... res.writeHead(200, {'Content-Type': 'text/plain'});
  ... res.write('hello world');
  ... res.end();}).listen(5678);
```

Обратите внимание, что в первой строке подключается модуль HTTP. Node использует небольшие детализированные пакеты, предоставляемые менеджером пакетов npm. В отсутствие аналога Base Class Library среда Node смогла развиваться быстро, поскольку большая часть ее функциональности сосредоточена в пакетах. При распространении приложения подключаются только необходимые библиотеки. Эти детализированные модули отчасти повлияли на модуляризацию среды .NET. .NET Core существенно зависит от менеджера пакетов NuGet. Предыдущие версии NuGet могли использоваться для загрузки общих библиотек — таких, как Autofac и Newtonsoft.Json.NET. Можно сказать, что из-за высокой детализации .NET Base Class Library каждый разработчик .NET должен иметь представление о программе NuGet.

Минималистичные проекты из мира Node также вдохновили разработчиков на создание облегченных проектов, генерируемых при проектировании нового проекта .NET Core. Этот подход был призван уменьшить размеры дистрибутивов и, теоретически, ускорить работу приложений. Он также минимизирует площадь поверхности носителя, подлежащую проверке, и потенциальную площадь поверхности, которая может быть подвержена атаке в случае возникновения пробелов в безопасности. В приложениях с высокой степенью модульности обнаружение уязвимости в одной части среды .NET потребует исправлений только в приложениях, использующих этот модуль напрямую (вместо перевыпуска всех приложений, построенных на базе .NET).

Среда Node повлияла не только на архитектуру .NET Core, но и на ASP.NET Core. Процесс построения ASP.NET Core по умолчанию также сильно зависит от Node.js в отношении компиляции ресурсов и даже управления пакетами на стороне клиента через bower (код написан на JavaScript и выполняется в Node.js). Пожалуй, правильнее будет сравнивать Node.js с .NET Core, нежели с ASP.NET Core. .NET Core отличается тем, что это не просто моноязыковая исполнительная среда; это еще и набор хорошо спроектированных и оптимизированных библиотек. О Node.js этого сказать нельзя, что приводит к массовой фрагментации в реализациях стандартных конструкций.

Angular и React

JavaScript — довольно популярный язык. Он прошел долгий путь с того времени, как был языком, предназначенным в основном для валидации введенных данных в формах на веб-странице. Двумя самыми популярными фреймворками в среде разработчиков JavaScript являются Angular и React. Angular — более масштабный фреймворк, который охватывает большую функциональность, чем React. В него входит реализация привязки данных и контроллера, маршрутизации и системы внедрения зависимостей. React по сути является только лишь движком представления. Впрочем, оба фреймворка поддерживают нестандартные теги и объявле-

ния. Нестандартные теги позволяют создавать компоненты, инкапсулирующие разметку, функциональность и даже информацию о стилях. Если ваш сайт содержит множественные экземпляры элемента управления, общая функциональность может быть выделена в компонент. React и Angular сами по себе развивают идеи предыдущих фреймворков. В начале 2000-х в Java поддерживалась концепция теглетов (taglets); возможно, есть и более ранние примеры.

Представьте выпадающий список с выбором даты из календаря — стандартный элемент управления на многих сайтах. Каждый раз, когда пользователю нужно ввести дату, на экране появляется удобный маленький календарь. HTML5 определяет элемент выбора даты, удовлетворяющий функциональным требованиям, но не каждый браузер поддерживает этот элемент, а его внешний вид различается в разных браузерах. В большинстве случаев лучше воспользоваться готовым календарным элементом управления, разработанным сторонней фирмой. Однако такие сторонние элементы управления обычно обладают широкой функциональностью (порой даже чрезмерно широкой), а список параметров конфигурации выглядит как страница некрологов в газете из «Игры престолов». Необходимость просматривать параметры конфигурации каждый раз, когда на сайт добавляется новый экземпляр, весьма неприятна, не говоря уже о «геркулесовых подвигах» по обновлению всех экземпляров при изменении любого параметра. Было бы намного удобнее извлечь всю эту функциональность в одно общее место.

И React и Angular предоставляют такую возможность при помощи уже упомянутых нестандартных тегов. Представьте, что вам приходится использовать следующий фрагмент:

```
<div class="calendar-container">
  <h4>Birth Date </h4>
  <input id="datepicker" style="background: blue" data-start-date="1904-01-01"
    data-end-date="2012-01-01" data-allow-range="false" data-date-format="yyyy-MM-dd"/>
</div>
```

Так как этот фрагмент содержит большое количество конфигурационных данных, которые легко забываются, можно просто использовать следующую запись:

```
<calendar name="birthDate" label="Birth Date"/>
```

В компоненте `calendar` объединяется вся конфигурация, необходимая для согласованного отображения календаря на всем сайте. В определении компонентной структуры можно зайти еще дальше, создав элемент управления для дней рождения:

```
<birth-date-calendar name="birthdate"/>
```

ASP.NET Core MVC использует такую идею применения нестандартных тегов HTML для описания компонентов. Эта тема намного подробнее раскрывается в главе 19 «Многоразовый код».

Как ни парадоксально, эти компоненты приближают ASP.NET Core MVC к модели пользовательских элементов управления, популярной в Web Forms. Это была одна из составляющих высокой эффективности Web Forms. Разработчик мог легко разместить на странице несколько элементов, а потом связать их воедино. Поддержка такой функциональности в современных веб-фреймворках позволяет воспользоваться преимуществами обеих областей: быстротой разработки приложений и простыми, стандартизированными компонентами.

Открытый код

Нельзя не поразиться культурному сдвигу в отделе разработки Microsoft в связи с переходом на модель открытого кода. Исходный код ASP.NET MVC был опубликован, так что любой разработчик может читать его и учиться на его примере. Публикация кода позволила разработчикам заглянуть в недра реализации, чтобы лучше понять и решить некоторые особенно сложные проблемы.

ASP.NET Core MVC развивает концепцию открытого кода в Microsoft еще дальше. Если исходная версия ASP.NET MVC была просто выпущена с полностью открытым кодом на CodePlex, разработка ASP.NET Core MVC велась на GitHub в открытых репозиториях, и различные проекты с открытым хостингом получали запросы от сообщества. В результате столь демократичного подхода появился продукт намного более впечатляющий, чем если бы он был просто написан группой ASP.NET в ходе внутренней разработки. Творческий вклад участников порой оказывался очень интересным. Если вы встретили какую-либо функциональность, отсутствующую в ASP.NET, или обнаружили ошибку — вместо того, чтобы просто дожидаться следующей версии, примите активное участие в проекте и отправьте свой код. И хотя нет никаких гарантий, что он будет принят командой разработчиков, вы точно получите обратную связь. Подход к разработке ASP.NET изменялся под влиянием бесчисленных проектов с открытым кодом. Рассмотрим некоторые из них.

OWIN

Работа над OWIN (Open Web Interface for .NET) велась несколько лет. Исходная версия ASP.NET определяла довольно большие объекты для запросов и ответов — достаточно большие, чтобы они разрушительно сказывались на процессе модульного тестирования. Кроме того, эти объекты были сильно связаны с IIS, так как заполнялись конвейером обработки запросов IIS. OWIN определяет открытый интерфейс, который может использоваться между веб-сервером и ASP.NET. Соблюдение этого стандарта позволяло предоставлять страницы ASP.NET другим серверам, отличным от IIS. Первоначально проект OWIN был инициативой сообщества, но впоследствии он был всецело принят командой ASP.NET.

По мере становления OWIN и ASP.NET синергия между ними дошла до такого состояния, что в новый проект, созданный в Visual Studio, включались библиотеки OWIN. Кроме того, интерфейс OWIN стал официальным механизмом обмена данными между ASP.NET и выбранным вами веб-сервером. Монокультурность IIS была нарушена. Kestrel, веб-сервер на базе libuv, — пример сервера, набирающего популярность в мире ASP.NET Core благодаря поддержке Linux.

Также следует заметить, что OWIN — стандарт, а не реализация. За прошедшие годы появилось несколько разных реализаций, таких как Katana и Freya. ASP.NET Core — еще одна реализация.

Мы посвятили OWIN собственный раздел, но эту тему с таким же успехом можно было бы включить в раздел «Rails». Дело в том, что несколько лет назад в Rails была предпринята очень похожая попытка определения интерфейса между веб-серверами и Rails. В результате появился интерфейс Rack, обеспечивающий расширенные возможности взаимодействия между веб-серверами и различными веб-фреймворками Ruby (Rails — всего лишь один пример такого фреймворка). Также этот материал можно было привести в разделе «Node.js». У Node имеется промежуточная прослойка connect, из которой были позаимствованы многие идеи OWIN.

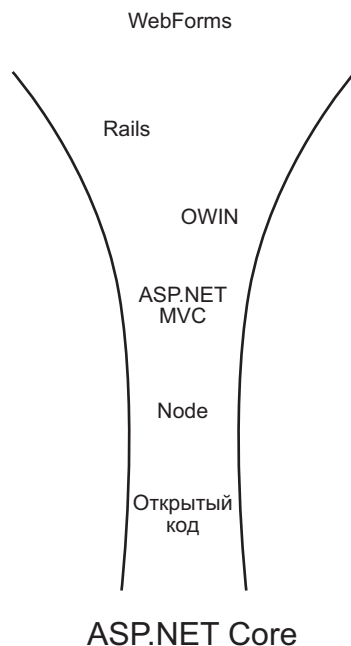


Рис. 2.2. Факторы, повлиявшие на становление ASP.NET Core

Объединение лучших аспектов разнообразных технологий (рис. 2.2) — действительно выдающееся достижение. Каждая концепция была адаптирована в стиле .NET. Конечный результат этих усилий вовсе не кажется набором разрозненных частей; все составляющие отлично сцепляются друг с другом. Самое потрясающее, что такой результат был достигнут за столь короткий срок; похоже, нас ждет интересное будущее.

Итоги

Может показаться, что ASP.NET Core — совершенно новый фреймворк, однако в нем легко прослеживается влияние многих других технологий. В результате появился симпатичный гибрид передовых решений и знакомых методов. Разработчики практически из любого сообщества найдут в ASP.NET Core что-то знакомое.

В следующей главе рассматриваются основные структурные элементы ASP.NET Core MVC: модель, представление и контроллер.

3

Модели, представления и контроллеры

Появление Адриана возле рабочего стола Даниэль было неожиданным. Возможно, еще большей неожиданностью стал его озабоченный вид. «У тебя найдется минутка?» — спросил он негромко и ушел, не дожидаясь ответа. Она озадаченно кивнула (в основном себе) и последовала за ним в соседнее помещение. Даниэль окинула взглядом комнату разработчиков, но никто не смотрел в ее сторону и тем более не проявлял ни малейшего интереса. Было совершенно непонятно, с чего Адриан вдруг стал изображать секретного агента.

Он прикрыл за ней дверь. «Послушай, я спец по CSS. Давай начистоту: я достаточно хорошо знаю jQuery, чтобы решать задачи, но я не программист. — Он держался довольно напряженно. — Похоже, все согласились с этим переходом на Core, или как там оно называется... но я... послушай, у меня ноутбук с Linux, и на нем даже нет Office». В комнате повисла тишина.

«Ты все еще переживаешь, Адриан? Марк-1 упоминал, что ты беспокоился по поводу сокращения». Откровенно говоря, Даниэль тоже переживала. В результате слияния она лишилась нескольких хороших друзей, но у нее не было уверенности, что сейчас стоит говорить об этом.

«Да, пожалуй, — ответил он. — Но я знаю об MVC только то, что это означает “модель-представление-контроллер”, и глубже пока не заглядывал. А все вокруг говорят, что это фреймворк. Если это фреймворк, но нам приходится самим создавать модели, представления и контроллеры, MVC в названии выглядит как-то странно, тебе так не кажется?»

В этом был резон. «Да, пожалуй, это правда», — сказала Даниэль.

«Я смотрю на все это под совершенно другим углом, и сейчас хочу просто понять, что происходит. Я готов учиться, но не знаю, с чего начать. — Адриан залпом проглотил остатки кофе так, словно боялся, что он испортится. — Да, нам говорили, что если мы еще здесь — то мы в безопасности. Но я не хочу выглядеть болваном, а то начальство решит, что группу можно еще чуть-чуть сократить».

«Понятно, — сказала Даниэль. — У меня сейчас есть немного времени, давай пройдемся по азам — это будет полезно нам обоим. Все будет хорошо».

М, V и C

Будем откровенны: MVC Framework — довольно скучное название. Сокращение, встречающееся в названии, известно по знаменитому паттерну «модель-представление-контроллер», который способствует структурированию проекта. Это название буквально выражает стремление к разделению обязанностей и отходит от другого распространенного паттерна того времени — «страница-контроллер». Название также может создать некоторую путаницу. Фреймворк — нечто много большее, чем модели, представления или контроллеры. В ASP.NET Core MVC входит непрерывно расширяющийся набор инструментов и библиотек, с помощью которых разработчик может создавать превосходные приложения на уровне современных веб-технологий.

Сейчас мы вкратце повторим аспекты, которые вам уже известны, а затем перейдем к более интересным аспектам фреймворка, официально известного как ASP.NET Core MVC.

Подробнее о моделях

На первом месте в аббревиатуре MVC стоит буква М (Models), поэтому мы начнем с модели. Модель представляет собой данные, которые необходимы для правильной визуализации взаимодействия (или его части) пользователя с сайтом. Пользователь приходит на страницу в приложении, управляемом данными, а модели являются частью данных. Однако в действительности именно модель должна обеспечивать поддержку визуализации представления, а не те сущности, которые сохраняются в базе данных.

Рассмотрим пример из возможной структуры базы данных приложения Alpine Ski House со сводной информацией о пользователе (рис. 3.1). Если вы хотите сообщить пользователю, что у него есть сезонный абонемент, было бы странно возвращать в представление список всех сезонных абонементов и перебирать содержимое коллекции для поиска действующего абонемента.

Возвращать всю эту информацию в представление не нужно. В листинге 3.1 приведена модель представления, которая более точно воспроизводит информацию, отображаемую для пользователя. Как видите, это обычный объект CLR, свойства которого используются для удовлетворения потребностей представления; при этом самому представлению не приходится принимать какие-либо решения относительно выбора отображаемой информации или реализации бизнес-логики. Представлению не обязательно знать, какой сезонный абонемент считается действующим, копаться в подробностях процесса покупки или перебирать дочерние записи из других таблиц, чтобы разобраться в данных.

Листинг 3.1. Класс AccountSummaryViewModel

```
public class AccountSummaryViewModel
{
    public Guid UserId { get; set; }
```

```

public int YearsOfMembership { get; set; }
public bool IsCurrentSeasonPassHolder { get; set; }
public List<string> IncludedFamilyMembers { get; set; }
}

```

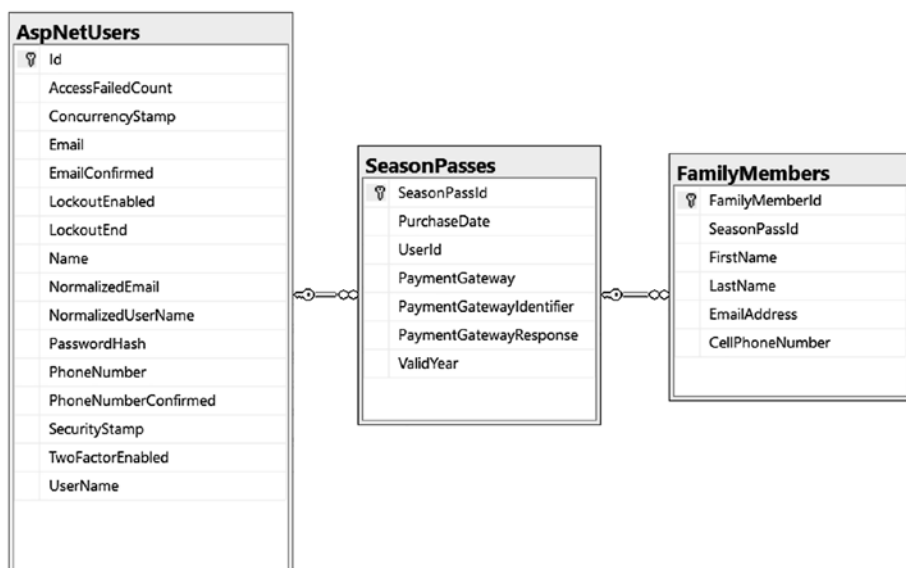


Рис. 3.1. Структура таблиц, которые могут использоваться для моделирования приобретения сезонных абонементов

Различия между построением моделей для интерфейсной части и хранением информации в базе данных важны не только для отделения обязанностей представления от бизнес-логики, заложенной в его основу, но и для предотвращения некоторых разновидностей проблем безопасности. Когда представление использует сущность из базы данных для записи, повышается вероятность того, что приложение пострадает от некорректной привязки или хакерских атак. Некорректная привязка (*overbinding*) происходит тогда, когда в форме или параметрах строки запроса присутствуют поля, не предполагавшиеся во входном запросе. Связыватель модели видит свойства, не знает, что их здесь быть не должно, и любезно заполняет данные за вас. Для примера рассмотрим класс, представляющий некий цифровой ресурс (листинг 3.2).

Листинг 3.2. Класс AccountSummaryViewModel

```

public class DigitalAsset
{
    public Guid AssetId { get; set; }
    public Guid AssetOwnerId { get; set; }
    public string Title { get; set; }
}

```

```
public string Description { get; set; }  
public Uri AccessUri { get; set; }  
}
```

Такая модель может использоваться для вывода списка ресурсов, доступных для пользователя; такой вариант в общем безвреден. Но если тот же объект будет использоваться для накопления изменений в записи, злоумышленник может использовать тот факт, что `AssetOwnerId` является свойством, и стать владельцем ресурса. Собственно, именно таким способом Егор Хомяков получил административные привилегии для репозитория Ruby on Rails (RoR) на GitHub в 2012 году¹. Уязвимость в RoR была использована через некорректные модели, использовавшие функцию массового присваивания (mass assignment) — аналог автоматической привязки модели в ASP.NET Core MVC. К счастью, намерения Хомякова были чистыми, и никакого вреда он не причинил. Впрочем, из этих инцидентов был извлечен урок. Сегодня в нашем распоряжении появилось много способов защитить себя; они будут рассмотрены в главе 13 «Идентификация, безопасность и управление правами». Пожалуй, самый простой способ заключается в использовании моделей, подходящих для выполняемой задачи.

В большинстве примеров моделей представлений сущность из базы данных обычно используется напрямую как тип модели для представления; однако такое решение усложняет другие аспекты разработки (например, тестирование) и не способствует разделению обязанностей в ваших контроллерах. Прямое использование сущности в представлении означает, что вы достигли нежелательного уровня сопряжения базы данных с представлением.

Модель должна содержать все необходимое для визуализации страницы после того, как вы разобрались с бизнес-логикой. Часто она содержит «плоское» представление денормализованной записи из нескольких таблиц базы данных. По этим причинам, а также с учетом предназначения объекта, который вы строите при создании «модели», вероятно, следует рассматривать его как «модель представления» из-за его тесных связей с представлением.

Представления

В этом разделе речь пойдет о представлениях (Views) — букве V в нашем любимом сокращении. Представления в ASP.NET Core MVC представляют собой файлы, используемые для чередования частей модели с разметкой HTML, необходимой для отображения желаемого пользовательского интерфейса. Создав новый проект на базе шаблона приложения по умолчанию, вы найдете все представления в папке **Views**; также возможно провести поиск в Solution Explorer по условию «.cshtml» — это расширение используется для представлений Razor.

¹ GitHub reinstates Russian who hacked site to expose law, John Leyden, March 5, 2012, <http://www.theregister.co.uk>.

Используя механизм представлений Razor и синтаксис, задействованный в нескольких последних итерациях MVC Framework, вы сможете без проблем переключаться между синтаксисом, используемым для управления последовательностью выполнения или обращения к модели или сервисным функциям, и разметкой, необходимой для генерирования HTML.

В листинге 3.3 мы создали неупорядоченный список со значениями из коллекции `IncludedFamilyMembers`, принадлежащей модели. Razor позволяет использовать в HTML встроенный код C# и довольно умно интерпретирует то, что получает от вас. Простого символа `@` достаточно, чтобы парсер понял, что вы переключаетесь на C#, а поскольку угловые скобки не могут использоваться в начале допустимой команды C#, парсер понимает, когда вы возвращаетесь к HTML. Механизм Razor более подробно рассматривается в главе 11 «Представления Razor».

Листинг 3.3. Пример смешения C# и HTML в синтаксисе Razor

```
<ul>
    @foreach (var familyMember in Model.IncludedFamilyMembers)
    {
        <li>@familyMember</li>
    }
</ul>
```

Частичные представления

Панели инструментов, покупательские корзины, части информационных панелей и другие аналогичные компоненты приложения часто встречаются на нескольких страницах или даже на всех страницах сайта. В соответствии с принципом DRY (Don't Repeat Yourself, то есть «не повторяйтесь») такие компоненты можно создавать с использованием частичных представлений (partial views), которые в свою очередь могут повторно использоваться из любой другой страницы.

Частичные представления обычно не воспроизводятся самостоятельно, а используются в сочетании с другими представлениями в вашем проекте. Скорее всего, первым местом, в котором вы столкнетесь с ними в любом приложении MVC, будет файл `_Layout.cshtml`, в котором представление использует частичные представления для отображения статуса входа. Также среди других популярных применений частичных представлений можно выделить отображение кнопок на панелях инструментов, сводки содержимого корзины (они обычно отображаются в верхней части страницы на сайтах интернет-магазинов) или боковые панели с данными, актуальными для текущей страницы.

В предыдущих версиях MVC Framework визуализация дочерних действий должна была производиться асинхронно, но те же идеи, которые сделали возможными частичные представления, теперь могут использоваться для построения компонентов представлений и их асинхронной обработки. Компоненты представлений более подробно рассматриваются в главе 19 «Многоразовый код»; они играют

важную роль в некоторых ситуациях для поддержания нормального быстродействия на сайте. Примерами такого рода служат сложные генерируемые представления и частичные представления, взаимодействующие с сервисами; они будут рассмотрены позже в этой главе.

Но прежде чем пользователь сможет получить вывод представления, а вы сможете загрузить любую модель в ядро представлений, необходимо немного поговорить о контроллерах в вашем проекте.

Контроллеры (...и действия!)

Контроллеры (Controllers) — «регулирующие» в приложениях MVC, которые следят за тем, чтобы правильные биты передавались в нужные места. Контроллеры обычно наследуют от базового класса `Controller`, но, если функциональность базового класса вам не нужна, вы также можете воспользоваться соглашением и завершить имя своего класса словом `Controller` — например, `SeasonPassController`.

По умолчанию предполагается, что ваши контроллеры размещаются в папке с именем `Controllers` в корне проекта. Сейчас это не является строго обязательным, потому что Core MVC сканирует проект с использованием соглашений об именах и наследовании, и все же такое размещение рекомендуется использовать для более четкой организации проекта. Тем самым вы упростите управление и сопровождение базового кода для других разработчиков (а возможно, и для вас самих через некоторое время).

Мы, разработчики программного обеспечения, используем контроллер в качестве контейнера для взаимосвязанных наборов обработчиков входящих запросов. Такие обработчики называются действиями (actions) и реализуются в виде методов класса контроллера. Каждый метод (или действие) может получать ноль или более параметров, которые автоматически заполняются на шаге связывания модели в ходе конвейерной обработки, если они представлены во входящем запросе.

Наша цель как авторов этих «регулирующих» — программировать контроллеры с использованием общепринятых практик. Основной обязанностью действия является обработка запроса, проверка входных параметров и создание соответствующего ответа.

Время от времени для этого требуется создавать или запрашивать экземпляр класса модели, или выдавать ответ на базе подходящего кода статуса HTTP. Старайтесь избегать включения бизнес-логики в контроллер; это обязанность модели или других компонентов, как и работа с данными или внешние вызовы из действий, которые должны быть частью сервисных функций приложения. Высокоуровневая схема изображена на рис. 3.2.

Может показаться, что внешнее определение этих сервисных функций усложняет архитектуру или же поднимает вопросы типа: «А кто создаст эти сервисы для

меня?» Отличный вопрос, на который будет дан ответ в разделе «Внедрение зависимостей» далее в этой главе.

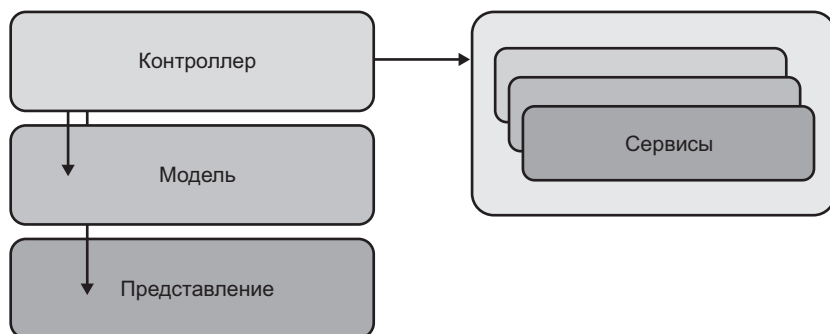


Рис. 3.2. Контроллер отвечает за активизацию бизнес-логики, задействованной в генерации ответа HTTP

Не только MVC

Как упоминалось ранее, ваши решения вовсе не ограничиваются связями между моделями, представлениями и контроллерами. Мы продолжим рассматривать эти темы в книге, но есть еще некоторые важные концепции, которые необходимо принимать во внимание в процессе разработки.

Промежуточное ПО

Откроем вам небольшой секрет об ASP.NET Core MVC: в сущности, всё это промежуточное ПО. Всё!

При запуске приложения у вас имеется возможность загрузить свою конфигурацию, настроить сервисы, а затем настроить конвейер обработки запросов, где вступает в игру концепция промежуточного ПО. В этом можно убедиться в методе `Configure` класса `Startup` шаблона проекта по умолчанию.

Часто описание промежуточного ПО и интерпретация его читателем чрезмерно усложняют относительно простую идею. Промежуточное ПО нужно для того, чтобы приложение могло сказать: «Каждый запрос должен обрабатываться следующими компонентами в указанном мной порядке». Промежуточное ПО — упрощенный вариант предыдущих реализаций подобной функциональности в ASP.NET, а именно обработчиков и модулей HTTP. Промежуточное ПО заменяет обе концепции способом, упрощающим реализацию.

Существуют несколько открыто распространяемых блоков промежуточного ПО для многих ситуаций, которые должны обрабатываться в ходе выполнения ва-

шего приложения как в служебных условиях (обкатка и контроль качества), так и в ходе реальной эксплуатации:

- *Диагностика*: обеспечивает обработку исключений и вспомогательные средства времени выполнения (такие, как страницы ошибок при работе с базой данных и выдачу технической информации для разработчиков).
- *Статические файлы*: обеспечивает укороченное выполнение конвейера обработки запроса для получения файла с диска.
- *Идентификация и аутентификация*: предоставляет приложениям средства для защиты конечных точек и ресурсов приложения.
- *Маршрутизация*: выбирает контроллер и выполняемое действие на основании входного пути и параметров.
- *CORS*: обеспечивает внедрение необходимых заголовков для совместного использования ресурсов между разными источниками (Cross-Origin Resource Sharing).
- *MVC*: обычно располагается в конце настроенного конвейера промежуточного ПО из-за поглощения запросов.

Каждый компонент промежуточного ПО имеет возможность выполнить код до и после следующего компонента в цепочке или же сократить процесс выполнения и вернуть ответ. Вероятно, название «промежуточное ПО» происходит от идеи о том, что фрагмент кода может выполняться в середине чего-то другого, как показано на рис. 3.3. В этом примере изображена серия разных запросов, которые обрабатываются различными компонентами промежуточного ПО в шаблоне проекта по умолчанию. В некоторых случаях запрос обрабатывается промежуточным ПО, использующим статические файлы и возвращающим ресурсы из `wwwroot`.

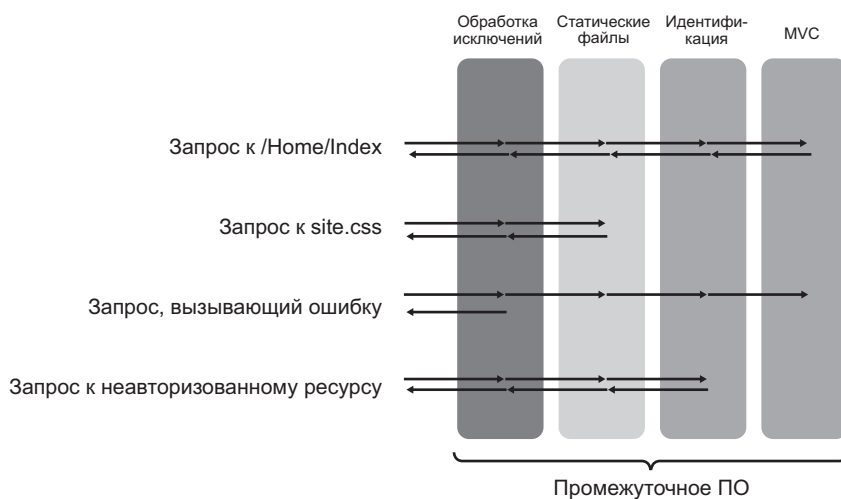


Рис. 3.3. Примеры обработки различных типов запросов промежуточным ПО

В других случаях запрос проходит весь путь по исполнительному конвейеру MVC, с созданием контроллера и возвращением представления.

Вы можете задействовать промежуточное ПО от сторонних разработчиков, другие вспомогательные компоненты от Microsoft или же написать собственный код для обработки сквозной функциональности, используемой в разных частях приложения. Конвейер промежуточного ПО также может разветвляться в зависимости от путей или предикатов, что позволяет создавать динамические, гибкие правила обработки запросов.

Внедрение зависимостей

О внедрении зависимостей написано много книг, но для полноты картины мы повторим основные положения.

В общем случае желательно, чтобы ваш код четко и однозначно выражал свои зависимости. В C# необходимые компоненты и сервисы размещаются в конструкторе, чтобы они всегда предоставлялись при создании экземпляра.

Рассмотрим конструктор класса `HomeController` (листинг 3.4). Класс требует, чтобы при создании экземпляра всегда предоставлялась реализация `ILogger`.

Листинг 3.4. Конструктор класса `HomeController`

```
public class HomeController{
    ILogger _logger
    public HomeController (ILogger logger)
    {
        _logger = logger;
    }
}
```

Классу `HomeController` не нужно знать, как настраивать или создавать реализацию `ILogger`; не нужно знать, где сохранять журнальные данные и как это делать. Но в любой момент после создания экземпляра `HomeController` сможет добавить полезную информацию в журнальные файлы, когда потребуется. Один простой параметр конструктора явно выражает требования; это называется *принципом явного выражения зависимостей*.

Чтобы этот контроллер был создан конвейером, в исполнительной среде должен присутствовать компонент, способный разрешить требование `ILogger`. Эти сервисные функции и компоненты настраиваются в контейнере, после чего соответствующие зависимости внедряются в конструкторы во время выполнения. И вот пожалуйста — внедрение зависимостей! Но поскольку эта тема более обширна, а также из-за того, что ASP.NET Core MVC вводит в область внедрения зависимостей ряд новых идей, идея инверсии управления будет более подробно представлена в главе 14 «Внедрение зависимостей»; там же мы выясним, что необходимо для замены контейнера по умолчанию.

Другие усовершенствования

ASP.NET Core MVC содержит и другие усовершенствования по сравнению с предыдущими версиями, и мы рассмотрим их в этой книге.

- *Конфигурация и ведение журнала*: эти критические аспекты приложений, которые явно запоздали с появлением в пространстве .NET, были переработаны, упрощены и получили полноправный статус. О них можно подробнее прочитать в главе 12 «Конфигурация и журналирование».
- *Тег-хелперы*: наряду с другими аспектами, упрощающими разработку интерфейсной части, в главе 19 «Многоразовый код» будут рассмотрены тег-хелперы и их сходство с разметкой HTML, которую мы пытаемся вернуть клиенту.
- *Удостоверения*: пользователь — не просто комбинация из имени и пароля. В главе 13 «Удостоверения, безопасность и управление правами» мы рассмотрим новые возможности, применения и компоненты, относящиеся к безопасности и управлению правами в ASP.NET Core MVC.

Итоги

Каждое очередное воплощение MVC Framework помогало сформировать какую-либо часть той технологии, которую мы видим сегодня. Некоторые уроки, усвоенные на этом пути, помогли предоставить более совершенные средства в распоряжение разработчиков, а на работу с моделями, представлениями и контроллерами, занимающими центральное место в наших приложениях, сегодня расходуется лишь небольшая часть усилий разработчика.

Они и не заметили, как подошло время обеда, а Даниэль уже успела разрисовать маркером всю доску. Адриан откинулся на спинку кресла и сказал: «Ничего себе, Даниэль... Да об этом можно написать целую книгу».

4

Структура проекта

Похоже, воодушевляющая речь Тима подошла к концу, и он немного запыхался. Если учесть, что его никогда не видели бегущим или хотя бы быстро идущим, подобная энергичность вызывала удивление. Чаще Тим казался чрезвычайно флегматичным и спокойно относился ко всему, что происходило в офисе. Он повалился в кресло и подал знак проворному молодому человеку в спортивной куртке и белой рубашке. «Балаш сейчас расскажет, чего мы хотим достичь за несколько следующих недель».

Балаш вышел на середину, от него так и веяло нервозностью. Похоже, он старался одеваться по книге «Мода для предпринимателей Кремниевой долины», а точнее, в соответствии с главой «Мода для технических директоров». Даниэль взглянула на Тима — тот утирал пот со лба. При всей внешней усталости он как-то умудрился прихватить пончик. Было совершенно неясно, как у него это получилось, потому что ближайшая коробка стояла в другом конце комнаты. Даниэль отогнала от себя мысль о том, не изобрел ли Тим некую тайную технологию телепортации пончиков, потому что Балаш уже взял маркер и направился к доске.

«Я три года провел в поисках решения задачи оптимизации подъемников и покупки билетов. Буду очень рад возможности применить свои идеи здесь. Наша компания установила золотой стандарт в обслуживании больших масс отдыхающих, и я думаю, мы сможем применить к этой ситуации многие наши наработки. Прежде всего нужно разобраться с покупкой абонементов. Как это делается сейчас?»

Балаш сделал небольшую паузу и обвел взглядом комнату. Он говорил с очень легким восточноевропейским акцентом, словно покинул те края в молодые годы.

Даниэль уже давно не покупала абонементы. Все работники получали годовые абонементы бесплатно. Так начальство пыталось вывести их за пределы офиса, чтобы они могли взаимодействовать с лыжниками и со склонами. Как сейчас покупают абонементы? Она не знала.

Чтобы избежать неудобных вопросов, она стала разглядывать что-то на полу, стараясь не встречаться глазами с Балашем. Оказалось, это было излишне — Балаш продолжал свою речь, не дожидаясь ответа.

«Я скажу вам, как это делается сейчас. Неэффективно. Я посмотрел цифры за прошлый год — более 98% абонементов продается прямо на склоне в выходные. 98% — да это почти 100%! Более 80% людей, покупающих абонементы на месте, используют кредитные карты, еще 16% — дебетовые карты. Наличные почти не используются, а один человек попытался расплатиться шкурой бобра. — Балаш фыркнул над собственной шуткой и продолжал: — При таком количестве электронных платежей у нас появляется возможность перенести покупку абонементов на более раннее время. Если мы предоставим возможность покупать абонементы заранее с компьютеров и смартфонов, можно будет сократить численность персонала в билетных кассах».

Даниэль решила, что идея неплохая. Никому не нравится стоять в очереди, а их клиенты были в основном людьми молодыми. Наверняка у них есть доступ к компьютерам, а у многих есть смартфоны. Балаш продолжал свой анализ: «Проблема в том, что мы продолжаем сканировать абонементы с использованием портативных считывателей штрих-кодов. Это медленно и ненадежно. Что еще хуже, мы печатаем абонементы на дорогой бумаге повышенной прочности и прикрепляем их к одежде пластиковыми хомутиками. Это дорого. Только производство такого абонемента обходится нам почти в доллар; нужно найти возможность сэкономить. Мы не можем предложить клиентам распечатывать билеты дома, потому что принтер есть не у каждого, и уж точно ни у кого нет бумаги, которая выживет после падения на лыжне, если она будет крепиться к одежде».

«Нам нужен новый механизм продажи билетов и управления билетами как на склоне, так и перед заходом на склон. Нам нужен проект Parsley. Позвольте рассказать вам, как это мне представляется».

Project Parsley — пример приложения, над которым мы будем работать в этой книге. Почти все примеры взяты из реального кода проекта Parsley, и вы можете загрузить полный проект с GitHub. Мы постарались по возможности приблизить разработку к реальности, поэтому в репозитории GitHub вы найдете сообщения о проблемах, код и полную историю закреплений.

ПРИМЕЧАНИЕ

Вам будет полезно покопаться в репозитории, чтобы увидеть, как изменялось наше мышление и как мы сами двигались по путям, описанным в книге. Репозиторий расположен по адресу <https://github.com/AspNetMonsters/AlpineSkiHouse>.

Приложение делится на несколько частей. Сначала нужно предоставить некое место, где клиент может ввести свою информацию для покупки абонемента. Затем следует каким-то образом сообщить системе о том, что абонемент был использован на склоне. Наконец, нужен административный интерфейс, через который бизнес-сторона может собрать информацию о том, что происходит и сколько абонементов было продано. Конечно, потребуется база данных для хранения всевозможной информации о клиентах и абонементах. В оставшейся части книги мы

покажем, как разрабатывалось приложение и какие технологические решения при этом принимались, а пока необходимо сформулировать, *что* должно делать приложение (а не то, *как* оно должно это делать).

Для многих разработчиков диалоговое окно **File ▶ New Project** становится песней сирены. «Нажми кнопку и создай новый проект, — зовет оно. — Не беспокойся, с бизнес-задачей разберемся в процессе программирования». О, сколько раз мы соблазнились этой песней и разбивались о скалы действия до обретения понимания. Гибкость — это очень здорово, но очень важно хотя бы в некоторой степени понять пространство задачи, прежде чем пускаться на создание нового проекта. Так что привяжите себя к мачте, как Одиссей, и проплывайте мимо зова сирен.

Лыжные склоны

Для читателей, незнакомых с горнолыжными склонами, начнем с азов. Катание состоит из двух частей: подъема и спуска. Спуск в основном обеспечивается силой тяжести и навыками катания на лыжах или сноуборде. Все трассы начинаются с места высадки, или верхней станции подъемника, и расходятся по разным направлениям по склонам горы. Трассы делятся на категории, обозначаемые значками: зеленый круг (простая), синий квадрат (средняя) и черный ромб (сложная), как показано на рис. 4.1. На некоторых горах существуют трассы с двумя черными ромбами — высшей категорией сложности. Довольно часто на такие трассы не распространяются меры защиты от лавин, поэтому для катания на них требуется специальное снаряжение (например, передатчик и лопата).



Рис. 4.1. Категории сложности

Подобно тому как трассы сходятся на вершине горы, они также снова сходятся у нижней станции подъемника. Это позволяет лыжникам двигаться по кругу: от подъемника на гору и снова к подъемнику. На многих горах установлено несколько подъемников, а на самых больших количество подъемников превышает 20, что позволяет им перевозить тысячи людей в час. За прошедшие годы появилось много разных типов подъемников, от простых буксирных канатов до более дорогих подвесных канатных дорог.

Многие горнолыжные курорты продают билеты на подъемник на бумажных листках, которые лыжники прикрепляют к своей куртке или штанам. У нижней станции подъемника абонементы обычно проверяет человек с портативным сканером

штрих-кодов. Абонементы обычно остаются действительными в течение всего дня, но вы также можете приобрести абонемент на полдня, на несколько дней и даже на год.

Абонементы продаются в киоске у основания горы. День катания обходится около \$50 — не так уж мало. Кроме того, от катания удовольствия будет меньше, чем от покупки лишнего экземпляра этой книги, так что не упустите шанс сделать правильный выбор.

Клиенты приезжают издалека, чтобы покататься на лыжах, и не желают проводить много времени в очереди на покупку абонемента. Это означает, что для привлечения лыжников на курорт необходимо предусмотреть возможность покупки абонементов через интернет.

К счастью, цена устройств радиочастотной идентификации (RFID, Radio Frequency Identification) в последние годы снизилась. Это позволило встраивать чипы RFID в карты. RFID — маленькие микросхемы, данные с которых можно считывать на небольшом расстоянии без физического контакта. Возможно, вы уже встречались с аналогичными технологиями (например, PayPass™ или PayWave™) в международных пластиковых картах или в картах-пропусках, используемых в офисных зданиях или отелях. С каждой картой связывается уникальный идентификатор, по которому карту можно отличить от карт-близнецов.

Скоро сканеры для RFID-карт займут место ручных сканеров штрих-кодов на канатных дорогах. Если у лыжника есть пропуск, он может пройти через проход к подъемнику. Если пропуска нет, проход остается закрытым. Кроме того, сканеры устанавливаются на вершинах некоторых трасс, чтобы собрать информацию о предпочтениях клиентов. На основе этой информации строятся карты популярности трасс. Она также позволяет лыжникам просмотреть историю посещений и определить, сколько времени им понадобилось для спуска. Вертикальное расстояние, пройденное за день, — полезная метрика для настоящих любителей горнолыжного спорта.

RFID-карта высылается по почте, если клиент купит ее заранее за время, достаточное для пересылки. Если времени на пересылку недостаточно или клиент предпочитает покупать абонемент «на месте», то карты также есть в наличии на склоне. Карты могут использоваться повторно, так что клиенту достаточно одной карты при разных посещениях в разные сезоны. И хотя исходные затраты на изготовление карт и рассылку выше, чем при печати бумажных абонементов, небольшая скидка убедит клиентов купить пластиковый абонемент и использовать его во время дальнейших посещений курорта.

При входе на сайт новому пользователю будет предложено ввести свои учетные данные и адрес доставки. Затем система запросит у пользователя количество «лыжников» или «клиентов», которые будут пользоваться картой.

По каждому клиенту достаточно ввести имя и возраст для определения стоимости; детские тарифы ниже взрослых. Вместо хранения информации о возрасте ру-

ководство желает хранить дату рождения, чтобы отдел маркетинга мог рассылать купоны на скидку ко дню рождения клиента.

После установления владельца карты пользователь может переходить к заполнению. Доступно несколько типов абонементов: однодневные, многодневные и годовые. Кроме того, поверхность горы делится на три зоны. Абонемент может действовать на одну или несколько зон. Каждый клиент может приобрести несколько абонементов, все купленные абонементы связываются с одной физической картой. Если у клиентов уже есть карты, дальнейших действий не потребуется. Система запоминает список клиентов для каждой учетной записи пользователя, а также связь клиента с физической картой.

Наконец, пользователю предложат ввести платежную информацию и завершить процесс оформления заказа. Это приведет к отправке физических карт в случае необходимости или мгновенной активации существующих карт, чтобы пользователь мог завершить покупку в машине по пути на горнолыжный курорт.

Когда клиент окажется у склона с картой, никаких дополнительных действий от него уже не потребуется. Его карта активирована, он может сесть на первый свободный подъемник и порадоваться тому, что избавлен от утомительного стояния в очереди.

И хотя пока речь идет лишь о базовом прототипе, опыт взаимодействия пользователя с сайтом должен быть по возможности приятным. Прежде всего необходимо принять решение по вопросу, который всегда вызывает жаркие споры — одностраничное или многостраничное приложение? Одностраничные приложения не перезагружают всю страницу при каждой загрузке. Вместо этого они перезагружают небольшую часть страницы — как правило, объединяя данные, загруженные с сервера в формате JSON, с шаблоном на стороне клиента. Многие фреймворки одностраничных приложений работают с моделью данных, представлением и контроллером способом, аналогичным ASP.NET MVP. Главное преимущество одностраничных приложений — большее удобство для пользователя. Так как приложению не приходится загружать всю разметку веб-страницы при каждом действии пользователя, приложение быстрее реагирует и вызывает меньше раздражения у пользователя. С другой стороны, одностраничные приложения интенсивно используют JavaScript, что все еще создает трудности для некоторых разработчиков. Разработка одностраничного приложения в прошлом была делом сложным и более продолжительным, чем создание нескольких страниц. Рост популярности JavaScript и почти безграничный выбор фреймворков для создания одностраничных приложений устранили все препятствия (или по крайней мере опустили их до уровня, при котором различия в создании одностраничных и традиционных многостраничных приложений становятся пренебрежимо малыми).

В духе свободы эксперимента, культивируемой в Alpine Ski House, клиентская часть приложения пишется с визуализацией представления на стороне сервера, а не на стороне клиента. Генерация HTML на стороне сервера может оказаться

более эффективной для некоторых устройств, для которых проще передать данные по сети, чем тратить вычислительные ресурсы на обработку кода JavaScript на стороне клиента. Возможно, также будет проще строить представления на стороне сервера, чем тратить значительные умственные усилия на написание разных частей приложения на разных языках. Даже если приложение генерируется на сервере, обойтись совсем без JavaScript все равно не удастся, так что не стоит полагать, что вы полностью избавлены от этой проблемы. Впрочем, необходимость в JavaScript все же существенно ниже, чем в одностраничных приложениях.

API

RFID-сканеры относительно дешевы и просты в настройке. Вымышленная модель сканера для этой книги — RT5500s — работает через HTTP. Сканеры подключаются к простому пропускному устройству, которое открывается на определенный промежуток времени при обнаружении сигнатуры RFID, распознаваемой как действительная. Чтобы проверить абонемент на действительность, сканер отправляет по сети сообщение сервису проверки. Этот сервис тоже придется написать, но спецификация очень проста.

Сканер отправляет физический идентификатор просканированной карты в формате JSON в запросе GET и ожидает получить обратно признак `true` или `false`. Собственно, этим взаимодействие и ограничивается. В стандартной реализации RT5500s не предусмотрена аутентификация, что теоретически может привести к проблемам. Например, какая-нибудь технически одаренная личность может отправлять запросы GET API, пока не обнаружит действительный идентификатор, после чего фальсифицирует его, чтобы бесплатно покататься на лыжах. Незащищенный «интернет вещей» (Internet of Things, IoT) станет серьезной проблемой в ближайшие годы. Уже сегодня многие пользователи покупают камеры с поддержкой Wi-Fi и оставляют их без защиты, из-за чего злоумышленники могут подключаться к камерам через интернет и следить за тем, чем люди занимаются в своих собственных домах. Приведет ли это решение нашей группы разработки к неприятным последствиям в будущем? Подождем и посмотрим.

Практически в каждом существующем веб-фреймворке предусмотрен простой механизм создания REST-совместимых сервисов. Следовательно, API оказывает минимальное влияние на выбор технологии. Более того, вы даже можете рассматривать API как микросервис, в основном независимый. Это позволяет вам реализовать его на базе любого языка или выбранной вами технологии. Даже если вы ошибетесь с выбором технологии в таком маленьком проекте, это не будет иметь катастрофических последствий. При небольшом размере пакетов объем переработки в случае возникновения проблем будет незначительным.

К сожалению, схема передачи данных — лишь самая простая часть реализации сканеров. С чем действительно может возникнуть сложность — это с правилами

определения действительности карты. Сначала необходимо просмотреть список абонементов для карты и найти в нем нужный абонемент. В любой момент времени активными могут быть сразу несколько абонементов, поэтому нужно выбрать последовательные правила выбора. Если абонемент является новым, его следует активировать. Наконец, необходимо записать информацию сканирования, относящуюся к абонементу. Безусловно, логика проверки абонемента станет одной из самых сложных частей приложения.

Административное представление

Кажется, Артур Кларк когда-то сказал:

«Любой достаточно сложной системе требуется административный интерфейс».

Следовательно, системе Alpine Ski House тоже потребуются инструменты администрирования. На данный момент нужно получать информацию о том, сколько людей отдыхает на курорте сегодня. Эта информация предоставляется страховым компаниям для проверки правильности уровня страхового покрытия. Кроме того, руководство желает получать отчеты о количестве людей, покупающих новые карты, для оценки успешности проекта.

За этим требованием стоит простое желание: понять, работает ли эксперимент. Курорт Alpine Ski House, живое воплощение концепции бережливого (lean) стартапа, хочет с минимальными затратами экспериментировать во время процесса разработки. Результаты таких экспериментов определяют, на что будут тратиться ресурсы разработки. Исходная версия административного представления в приложении имеет минимальную функциональность; однако она начнет расти, когда бизнес-сторона начнет понимать, какую аналитическую информацию можно получить при помощи приложения.

Административное приложение в основном предназначено для построения отчетов. Это означает, что в нем будет много красивых графиков и диаграмм. Начальство любит подобные вещи: высокоуровневые отчеты, на основании которых можно делать выводы. Фактически речь идет о панели с аналитической информацией.

Качество пользовательского интерфейса лишним не бывает, поэтому часть проекта Parsley, которую видит клиент, написана с использованием клиентского фреймворка, взаимодействующего с WebAPI. Считается, что такое решение способно обеспечить оптимальное качество взаимодействий без особого ущерба для эффективности. Также оно станет хорошей тестовой площадкой для перевода всего приложения на другое, менее распространенное решение. Важнее сразу выдать приложение для клиентов как есть, нежели доводить его до полностью завершенного состояния, — это еще одна составляющая менталитета бережливого стартапа.

Все вместе

В действительности мы строим не одно, а сразу три приложения: клиентскую часть, административную часть и API. Все части должны каким-то образом взаимодействовать друг с другом, так как они используют общие данные. Многие приложения такого рода строятся вокруг общей базы данных, которая обеспечивает интеграцию. Использование базы данных как общего ядра может создать некоторые трудности, если разные части попытаются выполнить запись в одни и те же таблицы. Изменение структуры в одной части приложения может нарушить работоспособность других частей. Проблемы возникают из-за трудностей с построением абстрактной прослойки над базой данных SQL. Обычно требуется создать хотя бы защитную прослойку, чтобы изменение модели данных не приводило к нарушению работоспособности приложения.

Наша модель данных делится на изолированные блоки, называемые *ограниченными контекстами* (bounded contexts). Термин «ограниченный контекст» позаимствован из мира предметно-ориентированного проектирования. Кратко изложить эту концепцию довольно сложно, но по сути речь идет о группировке данных и функциональности, изолированной от остальных частей приложения. Внутри контекста семантика имен остается неизменной. Звучит не слишком понятно? Возьмем концепцию клиента. Для отдела доставки клиент — это всего лишь адрес, тогда как для отдела маркетинга это набор предпочтений и атрибутов, определяющих маркетинговую политику по отношению к клиенту. Отдел доставки является ограниченным контекстом, как и отдел маркетинга. Обмен данными между этими ограниченными контекстами должен быть четко определенным и явно выраженным.

Защитная прослойка представляет собой абстракцию для обращения к ограниченному контексту, а конкретнее — при необходимости связать воедино несколько ограниченных контекстов. Эта функциональность часто реализуется с использованием паттерна «Адаптер» или «Фасад». Адаптер преобразует представление одного ограниченного контекста в другое. Например, и клиентское приложение, и API должны работать с таблицей, содержащей данные сканирования. Клиентское приложение читает данные, чтобы узнать, где клиент побывал в течение дня. API записывает в таблицу данные, и поэтому считается эталонным источником истинности данных сканирования. Это означает, что любые изменения в данных сканирования должны осуществляться через сервис API.

Если команда разработки API решит, что данные в таблице сканирования нужно перевести из местного часового пояса на время UTC, нам хотелось бы ограничить последствия такого изменения для потребителей данных (а именно клиентского приложения). Чтобы прочитать данные сканирования, клиентское приложение должно воспользоваться адаптером для преобразования данных в представление, на которое рассчитан код.

Адаптер может выглядеть примерно так:

```
public class ScanDataAdaptor{
    public Client.Scan Map(API.Scan scan){
        var mapped = Mapper.Map(scan);
        mapped.ScanDate = ConvertToLocalDate(mapper.ScanDate);
        return mapped;
    }
}
```

В этом случае адаптер используется для отображения полей предметно-ориентированного представления из API в предметно-ориентированное представление клиентского приложения. Наконец, новая дата в формате UTC конвертируется в дату местного часового пояса. Такие методы легко строятся и тестируются. Вы будете использовать защитные прослойки каждый раз, когда потребуется организовать передачу данных между ограниченными контекстами.

Определение предметной области

Предметная область проекта Parsley делится на несколько ограниченных контекстов. Диаграмма отношений между сущностями на рис. 4.2 также разделена на ограниченные контексты. Заметьте, что приложения, из которых складывается система, не имеют прямого соответствия с конкретными ограниченными контекстами. Мы разрешим своим приложениям взаимодействовать с несколькими ограниченными контекстами. Если бы мы стремились к использованию чистой предметно-ориентированной разработки или методологии микросервисов, возможно, взаимодействие одного приложения с несколькими ограниченными контекстами следовало бы запретить. В наших приложениях причин для беспокойства нет. Конечно, при этом защитная прослойка находится на своем месте и предотвращает случайное повреждение данных.

На диаграмме выделяются пять ограниченных контекстов. Начало — в левом верхнем углу, движемся по часовой стрелке; первым идет контекст входа. Он отвечает за аутентификацию и авторизацию пользователей в приложениях (клиентском и административном). Для этого используется стандартная функциональность ASP.NET Identity. Затем идет ограниченный контекст клиента; он содержит информацию о людях, которые катаются на лыжах или сноуборде. Мы сохраняем личную информацию о таких людях и об их картах. Далее идет ограниченный контекст местоположения, в котором отслеживается название географического расположения и название курорта. Эта информация используется для вычисления таких показателей, как вертикальное расстояние, пройденное за день (в метрах).

Предпоследний ограниченный контекст содержит информацию о типе абонементов. Как правило, тип абонемента представляет собой набор правил, определяющих действительность абонемента в определенный день на определенном курорте:

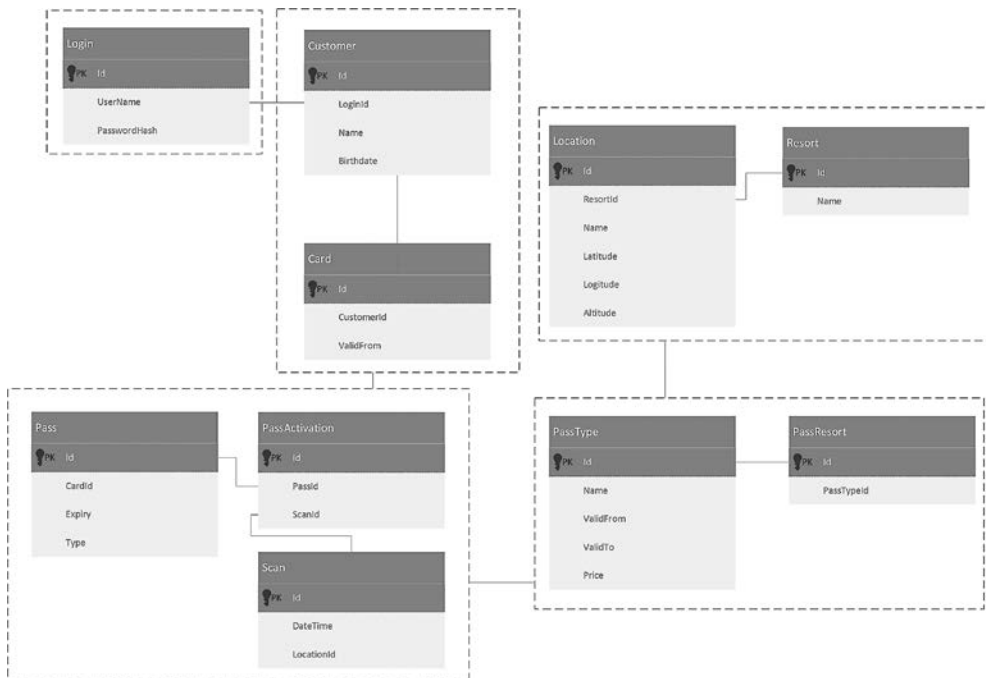


Рис. 4.2. Диаграмма отношений между сущностями с разбиением на ограниченные контексты

«трехдневный абонемент», «годовой абонемент» и т. д. Последний ограниченный контекст управляет распределением абонементов между картами и определяет, действительны ли они.

Итоги

В этой главе определяются обязанности трех приложений и базы данных. Нам предстоит проделать довольно большой объем работы за несколько спринтов. Тем не менее мы с полной уверенностью возьмемся за дело и построим великолепное решение. Вскоре клиенты смогут использовать свои новые карты и выходить на склон, минуя очереди в кассу. В главе 5 речь пойдет о том, как создаются приложения. Эта тема рассматривается так рано, чтобы первые версии можно было развертывать уже с первого дня.

5

Сборка кода

Прошла всего пара дней с начала первого спринта, а ситуация уже накалялась. Даниэль беспокоило, что отношения в группе никак не складывались. Участники грызлись из-за «битого» кода, когда Даниэль ошиблась с отправкой кода в репозиторий. Она добавила новый проект в кодовую базу, но забыла загрузить файл решения. Это была распространенная ошибка, которую мог допустить кто угодно, но команда уже была «на взводе», и каждая проблема казалась серьезнее предыдущей.

«Марк, — сказала она. — Я уже отправила файл проекта, так что сейчас все должно быть нормально».

«Да, нормально, но у нас целый час ушел впустую. Проект нужно запускать как можно быстрее. Упустить такой простой момент... У нас для этого совсем нет времени».

«Спокойно, — вступился Адриан. — Никто не идеален, все совершают ошибки. Просто нужно найти способ, который бы позволил избежать подобных проблем в будущем».

Только после этого в разговор вступил Балаш. «Вот и славно — я как раз ждал, пока кто-нибудь предложит что-нибудь позитивное вместо того, чтобы ругаться и жаловаться. Что мы можем сделать, чтобы предотвратить подобные проблемы в будущем? Чаще практиковать парное программирование?»

«Думаю, это поможет, — сказала Даниэль, — но на самом деле нам нужно другое — быстрое выявление ошибок. Нужно понять, как компенсировать различия в рабочих средах, чтобы никто не говорил, что на его машине все работает».

«Легко сказать, — ответил Балаш. — И как это сделать?»

«Можно создать билд-сервер и проводить на нем сборку каждый раз, когда в репозитории регистрируется изменение», — предложила Даниэль.

«Хорошо, — сказал Балаш. — Тогда за дело. Возвращаемся к работе».

Люди и компьютеры говорят на разных языках, и это понятно. Дело даже не в том, что компьютеры общаются на некоем эсперанто — они говорят на языке, который принципиально отличается от языков людей. Человеческие языки полны неоднозначностей, которые недопустимы при программировании. Компьютер имеет дело только с непреложными истинами и мыслит идеально логично. Если человек сможет понять предполагаемый смысл фразы «Я люблю готовить кошек и Лего», несмотря на пропущенную запятую, компьютер, к сожалению, решит, что кошки и Лего входят в меню. Человеку очень трудно мыслить на языке, который будет правильно понят компьютером, а из-за того, что компьютерные языки слишком «многословны», нам приходится использовать языки высокого уровня. Задача создания языка программирования, понятного для человека, сложна из-за необходимости выдержать баланс между лаконичностью и доступностью для понимания. Выбор языка — вопрос личных предпочтений, на который слишком часто влияют стандарты программирования в компаниях. Какой бы язык вы ни выбрали для написания приложений ASP.NET Core, код на языке высокого уровня неизбежно потребует перевода на язык, понятный компьютеру. Для этого код должен быть обработан компилятором. Если вы пишете код на VB.NET или C#, используйте компилятор Roslyn, а если вы пишете на языке F#, у него есть собственный компилятор.

Из этой главы вы узнаете, как выполнить сборку проекта ASP.NET Core, для чего создаются билд-серверы и как организовать конвейер сборки.

Сборка из командной строки

Многие программисты боятся командной строки. Возможно, «бояться» — неподходящее слово; они неуверенно себя чувствуют с командной строкой. По правде говоря, возможность сборки решения из командной строки очень полезна. У билд-серверов часто отсутствует пользовательский интерфейс, способный описывать сложные подробности процесса сборки, а ведь чтобы нарушить этот процесс, достаточно одного модального диалогового окна. Инструментарий командной строки в .NET Core пользуется заслуженным вниманием, а пользоваться им для сборки сейчас проще, чем когда-либо.

Выполнить сборку отдельного проекта .NET Core в командной строке ничуть не сложнее, чем выполнить команду на уровне проекта:

```
dotnet build
```

Результат выглядит примерно так:

```
Project AlpineSkiHouse.Web (.NETCoreApp,Version=v1.0) will be compiled because
inputs were
modified
```

```
Compiling AlpineSkiHouse.Web for .NETCoreApp,Version=v1.0
```

```
Compilation succeeded.
```

```
    0 Warning(s)
```

```
    0 Error(s)
```

```
Time elapsed 00:00:07.3356109
```

Все настолько просто. Если проект имеет внешние зависимости, возможно, вам придется сначала восстановить их командой:

```
dotnet restore
```

Команда обращается к любым определенным источникам Nuget и пытается восстановить пакеты, перечисленные в `project.json`. Результат выполнения команды выглядит примерно так:

```
dotnet restore
```

```
log  : Restoring packages for C:\code\AlpineSkiHouse\src\AlpineSkiHouse.Web\project.
```

```
      json...
```

```
log  : Installing Serilog.Sinks.File 3.0.0.
```

```
log  : Installing Serilog.Sinks.RollingFile 3.0.0.
```

```
log  : Installing Serilog.Formatting.Compact 1.0.0.
```

```
log  : Installing Serilog 2.2.0.
```

```
log  : Installing Serilog.Sinks.PeriodicBatching 2.0.0.
```

```
log  : Installing Serilog 2.0.0.
```

```
log  : Writing lock file to disk. Path: C:\code\AlpineSkiHouse\src\AlpineSkiHouse.Web\
```

```
project.lock.json
```

```
log  : C:\code\AlpineSkiHouse\src\AlpineSkiHouse.Web\project.json
```

```
log  : Restore completed in 27179ms.
```

Конечно, в наше время лишь немногие приложения достаточно малы, чтобы поместиться в одном проекте. О том, почему необходимы множественные проекты, рассказано в главе 24 «Организация кода». Если вы собираетесь выполнить сборку сразу нескольких проектов, потребуются более мощные средства.

ПРИМЕЧАНИЕ

Здесь и в ряде других мест книги упоминается файл `project.json`. Этот файл управляет тем, какие фреймворки поддерживаются вашими проектами, параметрами вывода, используемыми пакетами и дополнительными инструментами командной строки, предоставляемыми команде `dotnet`. Этот файл будет сильно изменяться в будущей версии .NET. Окончательная форма файла проекта еще не известна, но скорее всего, он снова перейдет на формат XML и снова будет называться `.csproj`. Также будет предоставлен механизм автоматического обновления, поэтому вам не придется беспокоиться об обновлении файла `project.json` до `.csproj`.

Многие годы стандартный подход к созданию множественных проектов основывался на создании файла решения в корне проекта. Файлы решений — очень простые файлы, которые фактически определяют набор конфигураций (например, **Debug** и **Release**), а также проекты, сборка которых должна выполняться для каждого профиля конфигурации. Допустим, конфигурация **Release** может включать только базовый проект без дополнительных инструментов, необходимых только в процессе отладки. Файлы решений поддерживаются как Visual Studio, так и MSBuild. Для сборки решения в командной строке используется следующая команда:

```
Msbuild.exe AlpineSki.sln
```

Команда перебирает все проекты в решении и выполняет для них команду сборки. В случае .NET Core используется команда `dotnet build`.

Если ваш проект работает с файлом MSBuild по умолчанию — вам повезло. Но если ваш проект слегка отличается от стандартного, стоит поискать альтернативные средства сборки. Один из возможных вариантов — PSake, программа сборки на базе PowerShell (<https://github.com/psake/psake>). Если же ваши интересы лежат в области функционального программирования, возможно, вам подойдет программа FAKE на базе F#. FAKE определяет предметно-ориентированный язык (DSL, Domain Specific Language) для настройки процесса сборки. Программа доступна по адресу <http://fsharp.github.io/FAKE/>.

Какой бы инструмент вы ни выбрали, скорее всего, в крупном проекте возникнет необходимость в выполнении дополнительных операций, выходящих за рамки простого компилирования проектов: сбора пакетов, выполнения тестов и отправки файлов на сервер символов (symbol server). Лучшее решение принимается по результатам обсуждения в команде; опробуйте некоторые инструменты и посмотрите, какой из них лучше подойдет вам.

Нашли? Отлично! А теперь установите его на своем билд-сервере. Ведь у вас есть билд-сервер, не так ли?

Билд-серверы

Важность билд-сервера невозможно переоценить. Частая сборка программных продуктов — отличный способ раннего выявления ошибок (желательно до того, как они доберутся до вашей пользовательской базы).

Вероятно, вам знакома крылатая фраза: «На моей машине это работает». Эта мантра часто повторяется в кругах программистов. На рабочих станциях разработчиков обычно установлено множество инструментов и библиотек, которые могут отсутствовать на компьютерах среднего пользователя. Слишком часто мы видим, что код работает на машине разработчика, но после развертывания в среде реальной эксплуатации он работать перестает. Никому не хочется оказаться в положе-

нии, когда приходится бормотать «А на моей машине работает», но из-за различий в рабочих средах это возможно.

Как избежать подобных неприятностей? Просто создайте билд-сервер, который компилирует и тестирует ваши программные продукты в другой среде. Даже в команде из одного человека быстрая обратная связь приносит неоценимую пользу при внесении изменений, нарушающих работоспособность. Если версия продукта работала час назад, а теперь она не работает, то винить в этом следует изменения, внесенные за последний час.

В больших командах постоянная интеграция с кодом коллег играет еще более важную роль. Например, если Даниэль отправляет код, конфликтующий с последними изменениями Марка-2, в идеале такие проблемы должны выявляться сразу же после возникновения, а не несколько недель спустя в процессе эксплуатации.

Кроме проведения непрерывных интеграционных сборок для проверки того, что внесенные изменения не создают конфликтов, также важно проводить ночные сборки. В рабочей среде или в пакете зависимости могут появиться изменения, «ломающие» блок программного кода, к которому никто не прикасался месяцами. Главный пример такого рода — модули Node. Хотя в проекте можно зафиксировать версию модуля Node, зафиксировать версии библиотек зависимостей не удастся. Такие изменения могут нарушить работоспособность вашего продукта, о чем вы даже не будете подозревать.

Мы используем термин «билд-серверы», но сервер способен на нечто много большее, чем простая сборка программного продукта. Билд-сервер должен отвечать за координацию целого конвейера сборки.

Конвейер сборки

Выпуск очередной версии программного продукта может быть весьма непростым делом. Многие компании стараются выпускать свои продукты ежемесячно, а иногда даже еженедельно или ежедневно. Такие компании, как Flickr, создают по несколько новых версий в день, потому что они уверены в своих процессах сборки и выпуска. Каждый раз, когда разработчик возвращает свой код в основную ветвь, запускается процесс сборки, и через несколько минут изменения разворачиваются на сайте. Это называется «непрерывным разворачиванием»: программный продукт постоянно находится в развернутом состоянии.

Главное преимущество такого подхода — сокращение времени от появления идеи до получения прибыли. Когда у бизнеса возникает новая идея относительно того, как изменить продукт и заработать, до реализации изменений не должно пройти слишком много времени. Непрерывное разворачивание также помогает выявлять дефекты безопасности. Вместо того чтобы ждать выпуска следующей версии по полтора месяца, все можно исправить и развернуть в течение дня.

Если эта цель кажется слишком амбициозной для вашей команды, возможно, для вашего стиля лучше подходит непрерывная поставка (continuous delivery). Методология непрерывной поставки находится всего в одном шаге от непрерывного развертывания; хотя вы можете использовать конвейер сборки и генерировать артефакты (а возможно, даже развертывать их в экспериментальной среде), построенные версии не отправляются в реальную эксплуатацию. При этом вы знаете, что каждая версия прошла тестирование и при необходимости может быть отправлена в эксплуатацию. Вам понадобится некоторое время на то, чтобы освоиться с процессом сборки и развертывания и убедиться в том, что в процессе развертывания человеческое вмешательство не требуется.

Конвейер сборки — так называется процесс перемещения кода, возвращенного в репозиторий, в некоторое итоговое состояние, будь то эксплуатация или простое развертывание в тестовой среде, пригодной для ручного тестирования. Как правило, конвейер сборки начинается с некоторого триггерного события — например, с возврата кода в систему управления версиями исходного кода. Пример конвейера развертывания представлен на рис. 5.1.

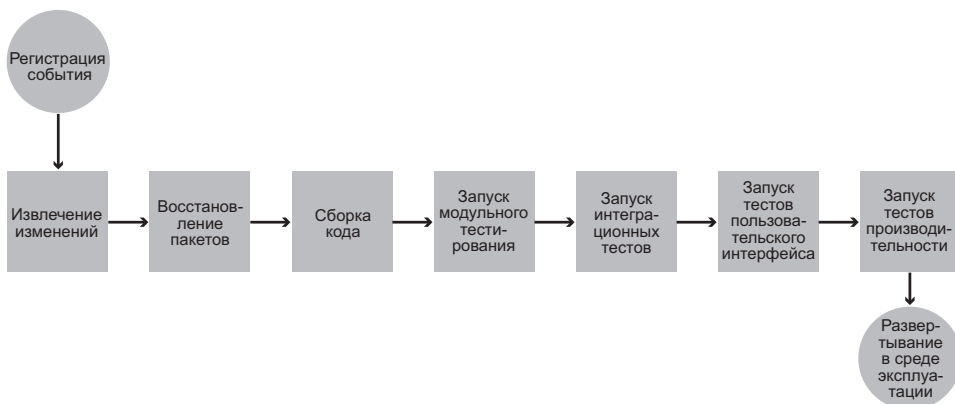


Рис. 5.1. Конвейер развертывания

Билд-сервер получает изменения с сервера управления версиями и запускает процесс преобразования кода в конечную форму. Типичным первым шагом становится восстановление пакетов. Для проекта ASP.NET Core восстановление пакетов, скорее всего, будет включать восстановление как пакетов nuget, так и пакетов Node. Возможно, в нем даже будут задействованы пакеты JSPM и bower. О пакетах JavaScript дополнительно рассказано в главе 15 «Роль JavaScript», а о nuget — в главе 16 «Управление зависимостями».

На следующем шаге производится сборка кода проекта. Вы уже видели, как это делается средствами командной строки. Далее выполняются тесты. Разные методы организации тестирования описаны в главе 20 «Тестирование». Пока достаточ-

но сказать, что существует много разных методов тестирования, и оно проводится в разных фазах. Однако нас интересует прежде всего быстрое выявление ошибок, поэтому сначала проводятся тесты, завершающиеся раньше всего, а это модульные тесты. После этого мы переходим к более затратным тестам — таким, как интеграционные и приемочные тесты.

ПРИМЕЧАНИЕ

В этой главе мы уже упоминали, насколько важно быстро выявлять ошибки. Почему, спросите вы? Потому что, если ошибки обнаруживаются на ранней стадии процесса, их исправление обходится дешевле, чем при позднем выявлении. Скажем, если проблема выявляется на рабочей станции разработчика, ее исправление обходится недорого, потому что она не отняла времени у других участников команды, так что потери компании сводятся к стоимости рабочего времени разработчика. Если же код доберется до клиента, затраты могут быть довольно значительными с учетом упущенных продаж, неверного ценообразования и других всевозможных проблем. Чем раньше будет выявлен и найден сбой, тем меньшие потери несет бизнес.

Существует стандартное правило: модульных тестов должно быть намного больше, чем интеграционных, а интеграционных — много больше, чем приемочных. Эти отношения представлены в пирамиде тестирования на рис. 5.2.



Рис. 5.2. Пирамида тестирования

В последнюю очередь выполняются тесты производительности. При развертывании новой функциональности очень важно убедиться в том, что это не приведет к потере производительности, с которой взаимодействие начнет раздражать пользователей или повредит стабильности системы в целом. В частности, компания GitLabs известна своей открытостью в отношении режимов развертывания и тестирования. В процессе развертывания они вычисляют сотни ключевых метрик и тщательно стараются улучшить их на стратегическом уровне.

График на рис. 5.3, позаимствованном из одной из недавних публикаций GitLab в блогах, показывает, как изменение в конфигурации памяти улучшает временные характеристики очередей HTTP.



Рис. 5.3. Временные характеристики очередей HTTP значительно улучшаются после изменений в конфигурации памяти

На рынке есть несколько превосходных вариантов билд-серверов. TeamCity — популярный вариант от JetBrains. Компания Atlassian, расширяющая свой спектр предложений по управлению жизненным циклом приложения, предлагает Bamboo. На стороне открытого кода есть Jenkins, а также несколько перспективных проектов Apache. Если вы не хотите запускать собственный билд-сервер, некоторые компании (такие, как CircleCI или Travis CI) выполняют сборку за вас на собственной инфраструктуре. Часто эта инфраструктура размещается в том же облаке, которое вы используете для размещения своего программного продукта, поэтому вам даже не придется использовать другой дата-центр.

В области инструментария сборки компания Microsoft предлагает TFS (Team Foundation Server). Существует как внутренняя версия TFS, так и версия с внешним размещением (хостингом). А теперь поподробнее разберемся в том, как организовать сборку Alpine Ski House в TFS.

Сборка приложения Alpine Ski House

Как упоминалось ранее, TFS предоставляет как версию с внешним размещением, так и внутреннюю версию. У Alpine Ski House нет времени на настройку билд-сервера на собственном оборудовании, поэтому была выбрана версия с внешним размещением.

В большинстве случаев этот подход предпочтителен. Кроме нескольких особых ситуаций, если ваша компания не специализируется на внешнем размещении серверов управления версиями, почему бы не доверить инфраструктурные хлопоты экспертам и не сосредоточиться на основной работе?

Первое, что вам потребуется, — учетная запись VSTS (Visual Studio Team Services). VSTS — версия почтенного сервера TFS (Team Foundation Server) с внешним размещением. Если ваша компания подписана на MSDN (Microsoft Developer Network), каждый разработчик имеет доступ к бесплатным минутам построения на VSTS. Даже после истечения этих минут расходы на покупку дополнительного времени минимальны. Агенты сборки (машины, на которых фактически выполняется сборка) подходят для большинства задач, потому что на них установлен широкий спектр инструментальных средств. Если у проекта есть особые требования (допустим, установка Oracle), вы можете предоставить собственных агентов сборки в облаке. Вы даже можете подключаться к агентам сборки, чтобы взаимодействовать с сервером VSTS. Это позволяет хранить сверхсекретную информацию внутри сети, при этом пользуясь преимуществами решения с внешним размещением. Приложению Alpine Ski House не нужен такой уровень безопасности, и в процессе сборки не происходит ничего необычного, так что агента с внешним размещением будет вполне достаточно.

Начните с создания в VSTS нового проекта для Alpine Ski House. Чтобы сделать это с основной информационной страницы, просто щелкните на кнопке **Add** под списком проектов. В диалоговом окне введите имя проекта и описание, как показано на рис. 5.4. Вам также будет предложено указать шаблон обработки. Если вы используете VSTS только для сборки, шаблон можно не указывать, потому что он управляет настройками истории и отслеживания неполадок в VSTS. Если вам захочется подробнее изучить тему шаблонов, на веб-сайте Microsoft есть достаточно документации по этому вопросу. Для управления версиями исходного кода выберите Git, хотя и это несущественно. Alpine Ski House использует для размещения GitHub, а не VSTS, а эта настройка только выбирает систему управления версиями для проекта. Если же вы используете VSTS, Git почти всегда оказывается лучшим вариантом, потому что этот стандарт получил широкое распространение в отрасли.

Create team project

Project name

Alpine Ski House

Description

A house in which we ski

Process template

Agile

This template is flexible and will work great for most teams using Agile planning methods, including those practicing Scrum.

Version control

Git

Git is a Distributed Version Control System (DVCS) that uses a local repository to track and version files. Changes are shared with other developers by pushing and pulling changes through a remote, shared repository.

Create project Cancel

Рис. 5.4. Создание нового проекта в VSTS

После того как проект будет создан, можно переходить к созданию сборки. Для этого следует выбрать команду **New** на вкладке **Builds**. Существует множество шаблонов сборки, на базе которых вы можете создать свое описание (рис. 5.5), но для наших целей хорошо подойдет вариант **Visual Studio** или пустой шаблон **Empty**.

Затем выбирается репозиторий, в котором будет храниться исходный код. На рис. 5.6 представлены разные варианты.

В диалоговом окне на рис. 5.6 удостоверения GitHub не настраиваются, но вскоре мы вернемся к этой теме. После того как новый проект сборки будет создан, вы попадаете на страницу описания сборки (рис. 5.7).

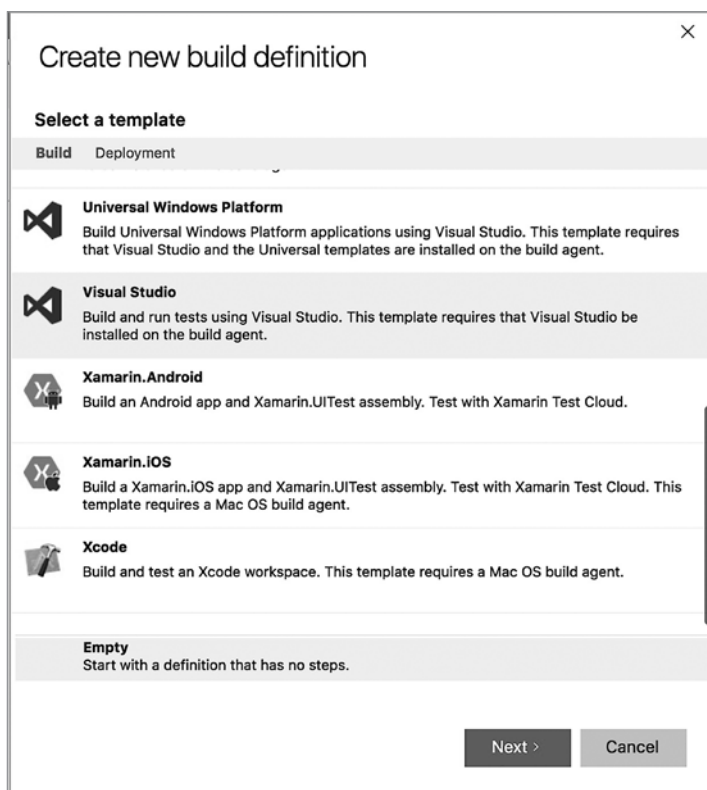


Рис. 5.5. Выбор шаблона сборки

Начнем с создания репозитория, чтобы система построения знала, где следует брать код. К счастью, между VSTS и GitHub существует отличная интеграция. Откройте панель управления, которая вызывается щелчком на значке с шестеренкой в правом верхнем углу страницы. На панели управления выберите **Services**, затем **New service endpoint** и выберите GitHub. В диалоговом окне на рис. 5.8 выберите либо личный маркер доступа (**Personal Access Token**), который можно взять из настроек GitHub, либо вариант **Grant Authorization** с последующей настройкой авторизации. В этом случае функциональность OAuth используется для авторизации VSTS для последующих действий от вашего имени на GitHub.

После того как вы свяжете VSTS с GitHub, вернитесь к описанию сборки и выберите вновь созданную учетную запись GitHub в качестве репозитория (рис. 5.9).

После завершения настройки управления исходным кодом можно переходить к настройке самого процесса сборки. Это можно сделать несколькими способами.

В данном случае мы выбираем простейший способ и пользуемся инструментами командной строки, которые уже были рассмотрены выше, вместо того чтобы полагаться на MSBuild. Пример процесса сборки показан на рис. 5.10.

Create new build definition

Settings
Repository source

AlpineSkiHouse Team Project | GitHub | Remote Git Repository | Subversion

☐ Continuous integration (build whenever this branch is updated)

Default agent queue | manage queues [?](#)
Hosted

Select folder
\\ Choose folder...

< Previous Create Cancel

Рис. 5.6. Выбор репозитория

Team Services / AlpineSkiHouse Simon Timms

HOME CODE WORK BUILD TEST RELEASE

Definitions Explorer

Build Definitions

Build Options Repository Variables Triggers General Retention History

Save

+ Add build step...

Рис. 5.7. Страница описания сборки

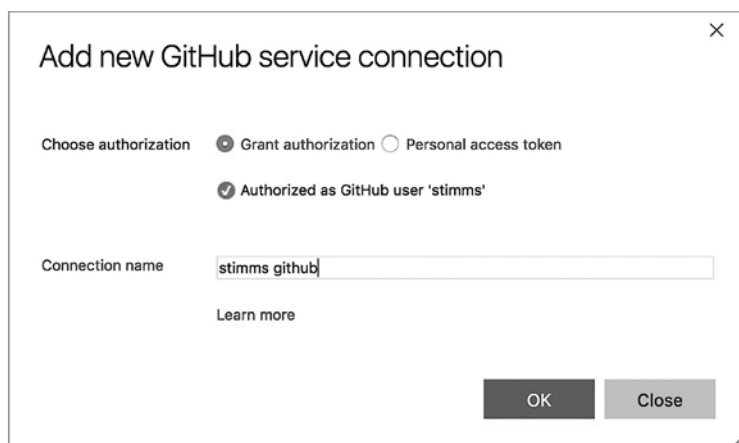


Рис. 5.8. Предоставление VSTS доступа к GitHub

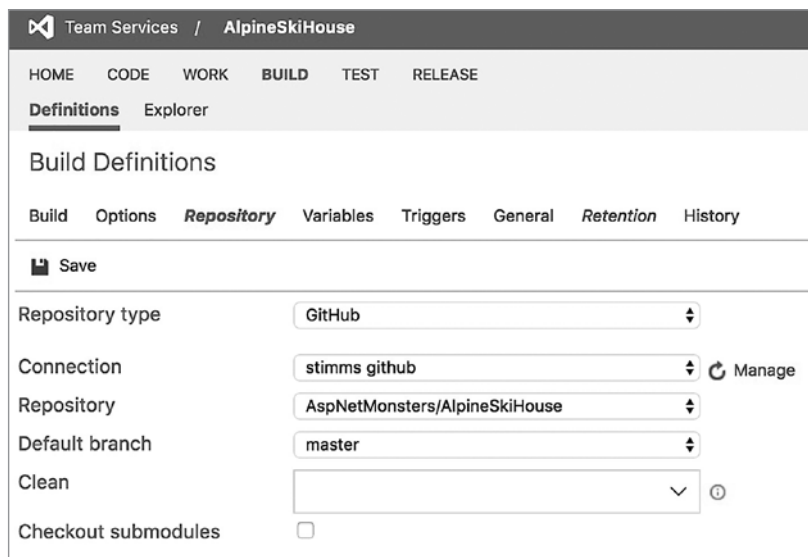


Рис. 5.9. Настройка репозитория в описании сборки

Первым шагом процесса сборки станет восстановление пакетов. В наше решение входит пара проектов, поэтому мы можем либо провести восстановление пакетов вручную для каждого проекта, либо написать сценарий для автоматического запуска восстановления пакетов. Если бы проектов было больше, безусловно, сле-

довало бы выбрать решение со сценарием. А пока группа задач просто запускает задачи восстановления пакетов для каждого проекта.

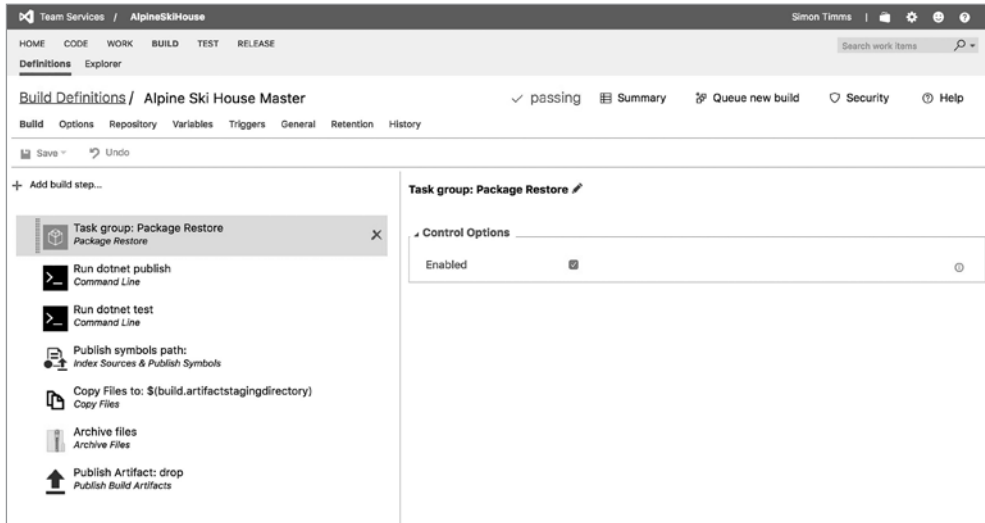


Рис. 5.10. Простое описание сборки

На следующем шаге выполняется команда `dotnet publish`. С файлом проекта по умолчанию она просто строит проект и перемещает файлы в каталог публикации. Впрочем, в нашем проекте в файле `project.json` определяется пара дополнительных задач:

```
"scripts": {
  "prepublish": [ "npm install", "bower install", "gulp clean", "gulp min" ],
  "postpublish": [ "dotnet publish-iis --publish-folder %publish:OutputPath%
                  --framework
                  %publish:FullTargetFramework%" ]
}
```

Эти задачи выполняются как часть этапа публикации. Обращения к `npm`, `bower` и `gulp` строят JavaScript и CSS для сайта. На шаге `postpublish` создается файл `web.config` для IIS.

Запуск тестов командой `dotnet test` будет рассмотрен в главе 20. Для Alpine Ski House на данный момент модульные тесты просты и эффективны.

На следующем шаге необходимые файлы перемещаются в каталог размещения (`staging directory`) с последующей упаковкой каталога размещения и публикаци-

ей артефактов. Далее эти файлы могут разворачиваться как часть процесса выпуска, кратко описанного в главе 6 «Развертывание».

Наконец, мы создаем триггер для запуска процесса сборки каждый раз, когда кто-то из участников вносит изменения в код. Как нетрудно догадаться, это делается на вкладке **Triggers** в VSTS. Как упоминалось ранее в этой главе, сборка должна выполняться и как часть непрерывного процесса проверки регистрируемого кода, и еженочно. Как видно из конфигурации на рис. 5.11, VSTS поддерживает такой сценарий.

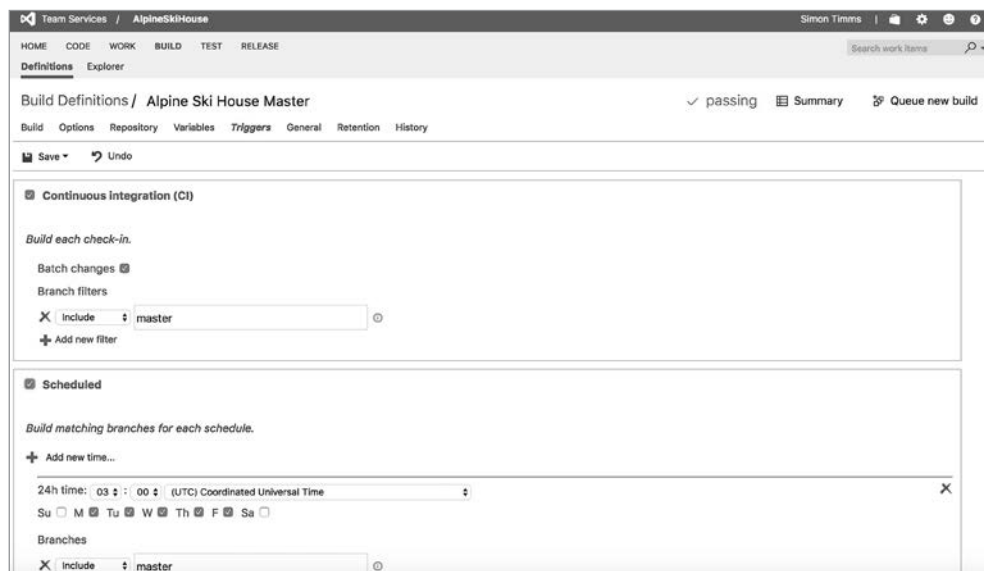


Рис. 5.11. Триггеры Alpine Ski House

Возможно, вы также захотите выполнить сборку ветвей, находящихся в главном репозитории. Теоретически эти ветви состоят в основном из запросов на включение изменений, так что их сборка до слияния повышает вероятность оперативного выявления проблем, пока они еще не начали замедлять работу всей команды.

С введением новой модели сборки система VSTS значительно улучшилась, и теперь она приблизилась к уровню таких продуктов, как TeamCity. Безусловно, выбранный нами простой подход к сборке с использованием только инструментов командной строки легко адаптируется для любой системы сборки. Количество параметров, с которыми можно экспериментировать при создании сборки, огромно, но выбранных нами частей достаточно для большинства проектов.

В конце концов, просто запустить билд-сервер и организовать успешную сборку — большой шаг для многих команд.

Итоги

В этой главе рассмотрена важность процесса сборки и его место в общем конвейере разработки. Эта информация позволила команде Alpine Ski House сократить трения при объединении кода. Частые сборки гарантируют, что проблемы будут выявляться на ранней стадии цикла разработки, а вы сможете очень быстро выдать последнюю версию своего программного продукта. Вся настройка сборок в VSTS была выполнена через серию диалоговых окон и подсказок, чтобы показать, как легко описывается процесс сборки с помощью современных инструментов. Все эти составляющие подготовили команду к успешному разворачиванию. В следующей главе мы займемся сохранением результатов сборки в облаке.

6

Развертывание

Балаш недолюбливал электронную почту, но был явно неравнодушен к импровизированным собраниям. Он называл их джем-сейшнами, как у джазменов. Похоже, другие тоже заинтересовались этой практикой, причем энтузиазм был настолько заразителен, что даже Даниэль быстро присоединилась к ним. По сути это было групповое программирование, при котором вся команда собиралась вместе и трудилась над одной задачей. Идея заключалась в том, что такая практика позволит уменьшить количество ошибок и сформировать малые группы для асинхронной работы над отдельными частями задачи. Если основная команда трудится над запросами к базе данных, другая команда в это время может разбираться с внедрением шифрования при подключениях к этой базе данных. Получив новые знания, вторая команда снова вливается в работу над проектом и вносит изменения в основной продукт. Таким образом предотвращаются простои при возникновении проблем, способные выбить из колеи единственного разработчика.

Сегодняшний «джем-сейшн» был посвящен развертыванию. Даниэль казалось, что пока еще слишком рано что-то развертывать. В сущности, у них была готова только страница лендинга. Больше пока ничего сделать не успели, но Балаш все объяснил. «Развертывание всегда откладывают на последнюю минуту, и люди в спешке занимаются им, когда время уже на исходе. Так появляются ошибки. Развертывание — сложная задача, поэтому давайте выполним его сейчас, и будем выполнять снова и снова, пока оно не станет простым».

«Почему оно станет проще, если его повторить много раз? — спросила Даниэль. — Развертывание — не отжимания; нельзя стать сильным от простого повторения».

«Ага, — сказал Балаш, — тебе нравится редактировать эти конфигурационные файлы вручную при каждом развертывании?»

«Ну уж нет!» — ответила Даниэль.

«Тогда почему бы не упростить себе жизнь и не написать сценарий, который будет это делать за тебя? Тогда процесс будет повторяемым, и тебе не нужно будет беспокоиться о возможных ошибках», — предложил Балаш.

Даниэль кивнула — ситуация прояснялась. «Если анализировать процесс каждый раз, когда мы его выполняем, и совершенствовать его, тогда все пойдет проще». Не совсем отжимания, но можно стать сильнее в развертывании за счет анализа своих действий. «Мне совершенно не хочется вручную подключаться к рабочим серверам для обновления пакетов».

«Теперь ты в теме, — воскликнул Балаш. Он всегда любил восточноевропейские поговорки. — За дело — наладим развертывание!»

В этой главе, посвященной развертыванию, мы настроим сервер Kestrel, объединив его с более полнофункциональным веб-сервером. Сборки проекта Parsley с предыдущего этапа будут развертываться и становиться доступными для всего мира. Сначала вы узнаете, как выполнять бинарное развертывание на компьютере с Linux; также будет рассмотрено использование Microsoft Azure. Мы создадим сайт, к которому можно будет обращаться для выполнения тестирования, а затем сделаем процесс развертывания таким простым и легко воспроизводимым, что он позволит создавать одноразовые тестовые среды для экспериментов с конкретными обновлениями и исправлениями.

Выбор веб-сервера

Долгие годы выбор веб-серверов для хостинга приложений ASP.NET был весьма ограниченным. Абсолютное большинство пользователей выбирало IIS — веб-сервер, входивший в поставку Windows. Некоторые администраторы запускали ASP.NET с Apache в Linux с использованием Mono, но это вряд ли можно назвать типичным решением. В последнее время появилась возможность использования веб-сервера, реализующего OWIN (Open Web Interface for .NET) — такого, как NancyHost.

Microsoft IIS — чрезвычайно эффективный и полнофункциональный веб-сервер. Он поддерживает изолированное размещение сайтов, что позволяет обеспечить хостинг многих сайтов на одном экземпляре сервера. Поддержка SSL, gzip, виртуального хостинга и даже HTTP2 — все это доступно сразу, «из коробки». Если вы развертываете веб-приложение ASP.NET в среде Windows, возможностей IIS почти всегда достаточно.

С другой стороны, .NET Core работает на разных платформах, на большинстве из которых IIS не поддерживается. В таких ситуациях выбирается Kestrel — облегченный веб-сервер, который поддерживает .NET Core и работает на всех основных платформах: Linux, MacOS и Windows. Центральное место в Kestrel занимает асинхронная IO библиотека libuv. Если название библиотеки кажется вам знакомым, то это потому, что эта же библиотека является «сердцем» Node.js.

Kestrel

Веб-приложения .NET Core представляют собой обычные исполняемые файлы. В предыдущих версиях ASP.NET конечным результатом компиляции обычно была библиотека, которая могла передаваться веб-серверу. Проект по умолчанию содержит файл `Program.cs` со следующим содержимым:

```
public static void Main(string[] args)
{
    var config = new ConfigurationBuilder()
        .AddCommandLine(args)
        .AddEnvironmentVariables(prefix: "ASPNETCORE_")
        .Build();

    var host = new WebHostBuilder()
        .UseConfiguration(config)
        .UseKestrel()
        .UseContentRoot(Directory.GetCurrentDirectory())
        .UseIISIntegration()
        .UseStartup<Startup>()
        .Build();
    host.Run();
}
```

Как видите, сначала создается конфигурация, которая читается из командной строки и окружения. Затем программа строит экземпляр `WebHost` и запускает его. Сначала мы добавляем хостинг для Kestrel и приказываем использовать текущий каталог в качестве корневого. Затем добавляется интеграция с IIS (так IIS взаимодействует с приложением). На последнем шаге конфигурации задается файл запуска, отвечающий за выполнение дополнительных настроек, специфических для сайта. Настройки приложения будут рассматриваться в главе 12 «Конфигурация и журналирование». Когда все параметры конфигурации хостинга будут определены, можно выполнить сборку конфигурации на хосте и запустить его.

Есть еще целый ряд параметров, настройку которых можно произвести на этом этапе. Например, директива `CaptureStartupErrors(true)` перехватывает исключения в процессе запуска и выдает страницу ошибок, вместо того чтобы просто завершить приложение. Это может быть полезно при разработке, но в продуктивной среде вариант аварийного завершения предпочтителен.

Параметры конфигурации хостинга не особенно многочисленны. Вся конфигурация хостинга хранится в точке входа приложения, чаще всего в `Program.cs`. Настройки самого приложения находятся в `Startup.cs`.

Kestrel — исключительно важный компонент в любой ситуации с хостингом приложений с ASP.NET, потому что именно он фактически выполняет код ASP.NET. До этой версии ASP.NET Core поддержка ASP.NET размещалась в рабочем про-

цессе IIS, `w3wp.exe`. В ASP.NET Core IIS работает как обратный прокси-сервер, отправляющий запросы Kestrel.

ПРИМЕЧАНИЕ

Эксперт по ASP.NET Рик Страл (Rich Strahl) опубликовал в своем блоге отличную статью со сравнительным анализом способов хостинга ASP.NET в IIS до настоящего времени и их отличиями от нового механизма ASP.NET Core. В статье также приводятся полезные советы относительно того, как избежать использования IIS в рабочем процессе разработки. <https://weblog.west-wind.com/posts/2016/Jun/06/Publishing-and-Running-ASPNET-Core-Applications-with-IIS>.

Обратный прокси-сервер

Чтобы понять, как IIS взаимодействует с Kestrel, важно понять, что именно делает обратный прокси-сервер. Прокси-серверы давно использовались для экономии ресурсов канала связи или ограничения сайтов, доступных для пользователя из сети (рис. 6.1).

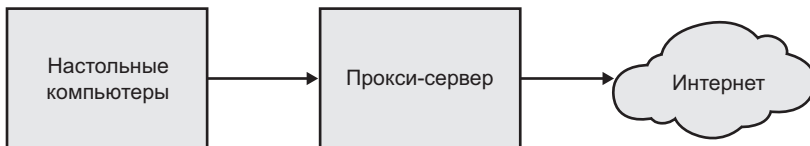


Рис. 6.1. Прокси-сервер

Прокси-сервер получает запросы HTTP и решает, являются ли они допустимыми. Если запрос относится к файлу, который был недавно просмотрен и кэширован, прокси-сервер предоставляет кэшированный файл, вместо того чтобы тратить ресурсы канала связи и создавать задержку при отправке запроса через интернет. Таким образом экономятся ресурсы канала и сокращается задержка при получении ответа пользователем. Интернет-провайдеры могут использовать этот механизм в своих центрах обработки данных для файлов, часто запрашиваемых пользователями. Например, при выходе нового трейлера популярного фильма прокси-сервер предотвращает излишнюю нагрузку на серверы в интернете. Обычно такое решение приносит пользу всем, хотя и создает сложности с обновлением кэшированных ресурсов при использовании распределенного кэша.

В схеме с обратным прокси-сервером (рис. 6.2) прокси-сервер находится перед веб-серверами. Как и в схеме с обычным прокси-сервером, этот прокси-сервер кэширует ресурсы. Запросы HTTP GET не изменяют данные на сервере, или по крайней мере не должны этого делать в соответствии со спецификацией, поэтому их контент может быть временно сохранен на прокси-сервере. Это снижает на-

грузку на веб-серверы, которым для обслуживания запроса обычно приходится проделывать гораздо большую работу, чем прокси-серверу.

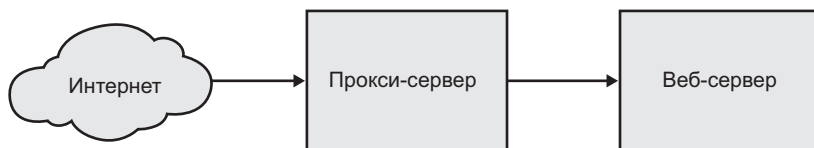


Рис. 6.2. Обратный прокси-сервер

Обратные прокси-серверы также могут выполнять функции абстрактной прослойки для группы веб-серверов. Прокси-сервер (рис. 6.3) располагается перед набором микросервисов, так что внешней стороне они представляются одним сервисом.



Рис. 6.3. Прокси-сервер может осуществлять маршрутизацию запросов между сервисами

IIS

Долгие годы сервер IIS был основным вариантом хостинга веб-сайтов на платформе Windows. ASP.NET Core поставляется с новым модулем IIS, который называется Asp Net Core Module. Этот платформенный модуль передает запросы от IIS напрямую серверу Kestrel, работающему в отдельном процессе, с использованием HTTP. Такая схема является интересным отклонением от метода, который использовался ранее. Интегрированный конвейер IIS оставался рекомендуемым решением в течение многих лет, но он очень жестко привязывает ASP.NET к IIS.

Чтобы технология ASP.NET могла работать на любой платформе, от этой привязки необходимо избавиться.

Чтобы включить IIS-хостинг для приложения ASP.NET Core, необходимо выполнить ряд условий. Первое — `UseIISIntegration`, о чем говорилось выше. Во-вторых, необходимо подключить файл `web.config`, загружающий `AspNetCoreModule`:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <!--
    Настройте параметры приложения в appsettings.json. Подробности https://go.microsoft.com/fwlink/?LinkId=786380
  -->
  <system.webServer>
    <handlers>
      <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModule"
resourceType="Unspecified"/>
    </handlers>
    <aspNetCore processPath="%LAUNCHER_PATH%" arguments="%LAUNCHER_ARGS%"
stdoutLogEnabled="false" stdoutLogFile=".\logs\stdout" forwardWindowsAuthToken=
      "false"/>
    </system.webServer>
  </configuration>
```

Как видно из этого примера, серверу IIS можно передать некоторые флаги. Один из самых интересных флагов определяет возможность передачи Kestrel маркеров аутентификации Windows. Это позволяет вашим приложениям Kestrel продолжать применение аутентификации NTLM в ситуациях с единым входом. Также указывается путь запуска при поступлении запросов.

`AspNetCoreModule` получает все запросы, обращенные к IIS, отправляет их Kestrel посредством HTTP, а затем возвращает клиенту то, что возвращает Kestrel. Схема изображена на рис. 6.4.

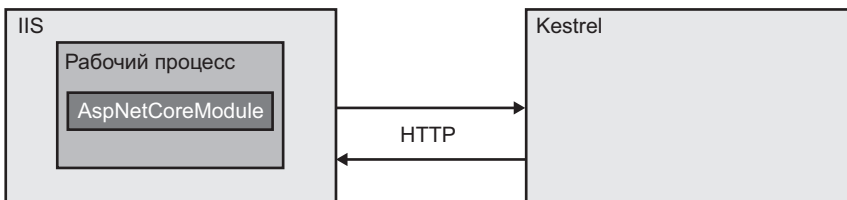


Рис. 6.4. IIS владеет рабочим процессом, в котором выполняется `AspNetCoreModule` для передачи запросов Kestrel

В файле `web.config` обработчик передает все запросы процессу Kestrel, но запросы могут фильтроваться, так что проходить через фильтр будут только запросы определенной категории. Например, регистрация обработчика может выглядеть так:

```
<handlers>
  <add name="aspNetCore" path="/new/*" verb="*" modules="AspNetCoreModule"
resourceType="Unspecified"/>
</handlers>
```

Обработчик пропускает только запросы, соответствующие пути `/new`. Это позволяет создавать гибридные приложения, у которых одни запросы обрабатываются ASP.NET Core, а другие — иным приложением.

ПРИМЕЧАНИЕ

Параметры конфигурации `AspNetCoreModule` хорошо документированы в официальной документации по адресу <https://docs.asp.net/en/latest/hosting/aspnet-core-module.html>. Обратите особое внимание на параметр `recycleOnFileChange`, который перезапускает службу Kestrel при изменении какого-либо из файлов зависимостей. Так обеспечивается ускоренное восстановление работоспособности сайта при развертывании.

Так как IIS уже не нужно обрабатывать управляемый код, возможно (и даже желательно) запретить управляемому коду веб-приложений в IIS использовать ASP.NET Core. Это должно обеспечить дальнейшее улучшение производительности.

Nginx

Nginx — высокопроизводительный веб-сервер, работающий в разных операционных системах, включая Windows, Linux и OSX. Также он предоставляет поддержку обратного прокси-сервера для ряда протоколов, включая MTP, POP3, IMAP и HTTP. Кроме того, Nginx может предоставлять статические файлы, обеспечивать завершение SSL и даже предоставлять поддержку HTTP2.

Nginx играет ту же роль, что и IIS, но для большего количества операционных систем. Вместо файла `web.config` вы можете создать файл `nginx.conf`, содержащий необходимые параметры конфигурации.

ПРИМЕЧАНИЕ

Более полный пример Nginx, включающий настройку SSL и HTTP2, можно найти на сайте ASP.NET Monsters по адресу: <http://aspnetmonsters.com/2016/07/2016-07-17-nginx/> и <http://aspnetmonsters.com/2016/08/2016-08-07-nginx2/>. Также вы можете найти видеоролики с тем же материалом на канале ASP.NET Monsters Channel 9 по адресу <https://channel9.msdn.com/Series/aspnetmonsters>.

Следующий конфигурационный файл настраивает простой прокси-сервер Nginx с Kestrel. Он также включает настройку завершения SSL. Так как веб-сервер Kestrel предельно облегчен, он не поддерживает SSL, а вместо этого полагается на то, что сервер будет выполнять завершение и отправлять незашифрованные запросы HTTP.

```

#количество порождаемых рабочих процессов
worker_processes 10;

#maximum number of connections
events {
    worker_connections 1024;
}

#предоставление информации http
http {
    #настройка типов mime
    include mime.types;
    default_type application/octet-stream;

    #настройка журналирования
    log_format main '$remote_addr - $remote_user [$time_local] "$request" '
        '$status $body_bytes_sent "$http_referer" '
        '"$http_user_agent" "$http_x_forwarded_for"';
    access_log /var/log/nginx/access.log main;

    #настройка сертификатов
    ssl_certificate/etc/letsencrypt/live/ski.alpineskihouse.com/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/ski.alpineskihouse.com/privkey.pem;

    #использовать sendfile(2) для прямой отправки файлов в сокет без буферизации
    sendfile on;

    #промежуток времени, в течение которого подключение остается активным
    #на сервере
    keepalive_timeout 65;

    gzip on;

    #настройка прослушивания
    server {
        #прослушивание порта 443 через http на ski.alpineskihouse.com
        listen 443 ssl;
        server_name ski.alpineskihouse.com;

        #настройка сертификатов SSL
        ssl_certificate cert.pem;
        ssl_certificate_key cert.key;

        ssl_session_cache shared:SSL:1m;
        ssl_session_timeout 5m;

        #запрет ненадежного шифрования
        ssl_ciphers HIGH:!aNULL:!MD5;
        ssl_prefer_server_ciphers on;

        #по умолчанию все запросы к / передаются localhost на порт 5000
        #это наш сервер Kestrel

```

```

location / {
    #использовать прокси-сервер для сохранения файлов
    proxy_cache aspnetcache;
    proxy_cache_valid 200 302 30s;
    proxy_cache_valid 404 10m;
    proxy_pass http://127.0.0.1:5000/;
}
}
}

```

В отличие от IIS, Nginx не запускает копию Kestrel за вас, поэтому вы должны независимо запускать их в рабочей среде. Для локальной разработки обычно нет необходимости использовать Nginx, если только ваша локальная разработка не требует функциональности, предоставляемой инфраструктурой Nginx. Считайте, что Nginx является чем-то вроде распределителя нагрузки в рабочей среде: это не тот компонент, который необходимо локально воспроизводить в большинстве тестов.

ПРИМЕЧАНИЕ

Несмотря на то что Nginx нормально работает в Windows, его взаимодействие с Windows не подвергалось значительной оптимизации. В результате в Windows по эффективности Nginx намного уступает IIS. Если в качестве среды размещения вы выбрали Windows, вашим основным веб-сервером останется IIS (по крайней мере в ближайшем будущем).

Публикация

Подход к публикации зависит от места назначения публикации и того, насколько у вас отработан этот процесс. В Visual Studio всегда существовала возможность создавать пакеты развертывания прямо на машине разработок. Обычно для этого разработчик щелкает правой кнопкой мыши на сайте и выбирает команду Publish (рис. 6.5).



Рис. 6.5. Список методов публикации

В открывшемся списке приведены различные варианты публикации, включая публикацию прямо в Azure, публикацию с использованием Web Deployment, FTP

или просто в каталоге файловой системы. Вероятно, из всех предложенных вариантов наиболее желательным является **Web Deploy**, который либо выполняет публикацию на удаленном сервере, либо собирает пакет для публикации на удаленном сервере с использованием программных инструментов. В параметрах прямой публикации можно указать комбинацию имени пользователя и пароля, которая используется для выполнения аутентификации с экземпляром IIS, поддерживающим расширения Web Deploy. Если вы управляете собственным экземпляром IIS, установите этот пакет (рис. 6.6), для того чтобы упростить публикацию пакетов.

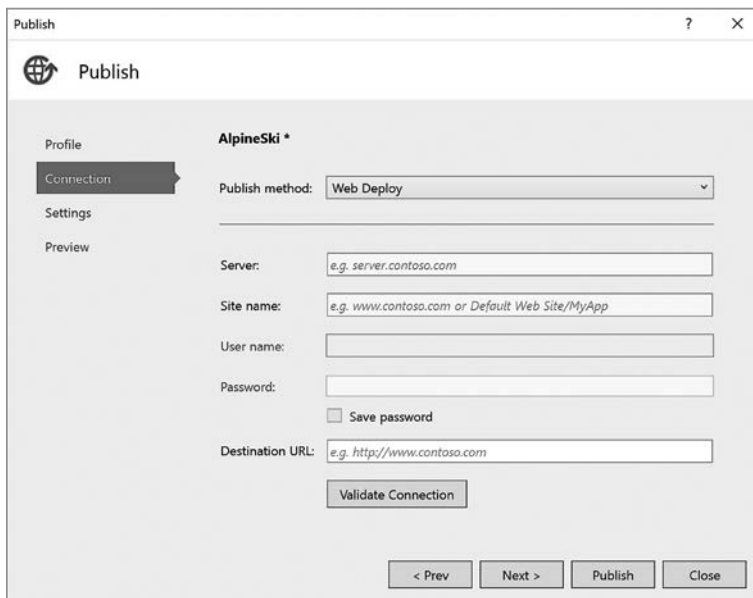


Рис. 6.6. Полное диалоговое окно Publish

Конечно, если вы публикуете свое приложение из Visual Studio, то делаете это совершенно напрасно. Публикация из Visual Studio ненадежна и плохо воспроизводима. Если вы уйдете в отпуск, а приложение должен будет публиковать кто-то другой, он либо не сможет это сделать, либо будет публиковать его с совершенно иными настройками. Развертывание по щелчку правой кнопкой мыши в Visual Studio — анахронизм, пережиток времен, когда мы не понимали, насколько важна воспроизводимость построения. Группа управления жизненным циклом приложения, консультирующая Microsoft по вопросам, связанным с передовыми методами DevOps, давно выступает за удаление функции развертывания из Visual Studio. К счастью, ведется значительная работа по созданию мощных и удобных инструментов — таких как TeamCity, Jenkins и TFS Build. Эти инструменты автоматизируют отслеживание изменений в репозитории, извлечение нового кода, его сборку и запуск тестов.

Если вы считаете, что их настройка создаст слишком много работы, существуют размещаемые решения (такие, как AppVeyor или Visual Studio Team Services), бесплатные для небольших групп разработки. Если у вас уже имеется лицензия MSDN, значит, у вас также есть лицензия Team Services, которая просто не используется эффективно.

Наличие централизованного билд-сервера позволяет всем участникам команды хорошо видеть текущее состояние сборок. Оно также помогает избежать конфликтов с кодом, который работает на вашей локальной машине, но с треском проваливается, когда его загружает другой участник команды. Централизованный билд-сервер также позволяет выполнять тесты в одном месте, что упрощает их создание и повышает надежность. Вам никогда не доводилось вносить внешне безобидное изменение, для скорости пропустить локальный запуск модульных тестов и узнать, что тесты перестали проходить у всей группы? Конечно, приходилось, потому что локальный запуск тестов бывает хлопотным делом. Со всеми такое бывало. Наличие централизованного билд-сервера гарантирует, что тесты будут выполнены, а ошибки и недочеты выявлены на ранней стадии.

Типы сборок

.NET Core может быть собран на базе полного фреймворка .NET либо на базе усеченной библиотеки .NET Standard Library. За последние годы было сделано много попыток создания версий .NET, работающих на разных платформах. Разумеется, существуют API, портирование которых на другие платформы бессмысленно. Например, вряд ли кому-нибудь потребуется изменять объекты Active Directory на Xbox. Предыдущим решением проблемы был PCLs (Portable Class Libraries) — набор целевых фреймворков (таких, как Silverlight или Windows Phone). При создании приложения вы определяете набор целевых фреймворков, а компилятор следит за тем, чтобы все используемые API входили в пересечение доступных API для перечисленных платформ. Впрочем, для использования PCLs устанавливались серьезные ограничения, особенно в отношении упаковки. Если целевой набор поддерживаемых платформ изменялся, приходилось перекомпилировать всю библиотеку и выпускать совершенно новые пакеты. Каждая комбинация целевых фреймворков идентифицировалась по номеру профиля. В любое обсуждение PCLs с большой вероятностью включаются числа (например, Profile136). В конечном итоге номера профилей создают невероятную путаницу без какой-либо пользы для обсуждения.

Технология .NET Standard разрабатывалась как способ решения тех же проблем, что и PCLs, но с меньшей путаницей. Вместо того чтобы определять набор поддерживаемых API в каждом целевом фреймворке, .NET Standard определяет базовое множество API, которое должно определяться любым фреймворком, претендующим на поддержку .NET Standard 1.0. Множество API в .NET Standard 1.0

достаточно мало, так как API должны поддерживаться везде, от Xbox до телефонов и фреймворков, работающих в Linux. .NET Standard определяет несколько наборов API со схемой нумерации, аналогичной нумерации версий. Каждый номер версии выше 1.0 поддерживает как минимум все API предыдущего уровня. Таким образом, .NET Standard 1.1 поддерживает все, что входит в .NET Standard 1.0, а также `System.Collections.Concurrent`, `System.Runtime.Numerics`, `System.Runtime.InteropServices` и ряд других API. В идеале ваше приложение должно ориентироваться на .NET Standard с минимальным номером версии.

Новое приложение в .NET Core в исходном состоянии настраивается для выполнения в портируемом режиме. Традиционно этот режим встраивается практически во все приложения. Конечным результатом сборки приложения является относительно небольшой набор файлов: созданная сборка (assembly) и сторонние библиотеки, необходимые для ее запуска. Предполагается, что все файлы, необходимые на стадии выполнения, уже существуют на целевом компьютере — как правило, они устанавливаются .NET Runtime Installer. В большинстве современных версий Windows имеется заранее встроенная версия .NET, хотя она может отличаться от версии, на которую рассчитано ваше приложение. В этом режиме файлы, полученные в результате сборки, являются архитектурно-нейтральными. При первом запуске приложения, развернутого в портируемом режиме, JIT-компилятор собирает и кэширует локальную версию нейтрального файла на машинном коде. В результате последующие запуски проходят намного быстрее, потому что они могут пропустить шаг компиляции.

Вместо сборки в портируемом режиме также используется построение в автономном (self-contained) режиме. В этой конфигурации приложение и все его зависимости компилируются в машинный код на стадии сборки, а не во время выполнения. Развертывание приложения автономного режима также включает все библиотеки, необходимые для запуска приложения, — даже библиотеки из фреймворка .NET. Выбор операционной системы и архитектуры целевой среды может показаться странным после многих лет работы в среде, поддерживавшей принцип «компилируется один раз, работает где угодно». Это необходимо, потому что реальная исполнительная среда .NET является приложением, откомпилированным в машинный код, — хотя в этом она отличается от работающих в ней приложений.

Одно из лучших новых качеств ASP.NET Core — полное бинарное развертывание. В предыдущих версиях ASP.NET на сервере должна была быть развернута правильная версия среды .NET. Такой подход создавал многочисленные конфликты между группами разработки, которые хотели иметь на сервере все самое новое и лучшее, и эксплуатационными группами, стремившимися к стабильности. .NET Core включает в развертывание всю исполнительную среду, что позволяет исключить риск установки потенциально конфликтующих сред .NET и нарушения работоспособности уже установленных приложений.

Сборка пакета

Начнем с упаковки приложения .NET в портируемом режиме. Так как эта конфигурация используется по умолчанию практически в каждом шаблоне, объем дополнительной работы минимален. Новый инструмент командной строки `dotnet` может использоваться для сборки каталога из приложения. Введите в командной строке в каталоге с файлом проекта следующую команду:

```
dotnet publish
```

Команда строит структуру каталога, необходимую для публикации приложения на сервере с уже установленным фреймворком .NET. Существуют некоторые рудиментарные параметры командной строки, которые могут передаваться на шаге публикации — режим сборки (обычно отладка или выпуск) и целевой каталог.

Этот каталог можно заархивировать и разместить там, где должна располагаться папка рабочего приложения. Если вы используете программу развертывания (такую, как Octopus Deploy), выходной каталог можно легко собрать в пакет Nuget. Пакеты Nuget обычно используются в качестве механизма развертывания по умолчанию для библиотеки .NET Standard. Чтобы упростить задачу создания этих пакетов, существует встроенный механизм сборки пакетов Nuget прямо из проекта. И снова для этого используется программа командной строки `dotnet`:

```
dotnet pack
```

Конечным результатом этой сборки является пара файлов `.nugget`. Они содержат двоичные файлы, сгенерированные при сборке, и символические имена, которые должны отправляться на сервер символов для отладки. Эти пакеты распространяются точно так же, как любые другие пакеты Nuget: они отправляются на сервер Octopus или распространяются через сайт (такой, как *nugget.org* или MyGet).

ПРИМЕЧАНИЕ

Если вы в своей работе предпочитаете использовать визуальные инструменты для сборки и упаковки своих приложений, возможно, вас удивит, что мы уделяем так много внимания инструментам командной строки. Дело в том, что все автоматизированные инструменты сборки и все инструменты визуального проектирования в действительности просто вызывают инструменты командной строки. А значит, вам будет полезно знать, что же при этом происходит «за кулисами».

Если ваш проект может использовать автономные пакеты, то в шаги сборки и упаковки необходимо внести некоторые изменения. Сначала нужно обновить файл `project.json`. Уделите следующую строку из секции `frameworks` для `netcoreapp 1.0`:

```
"type": "platform",
```

Затем необходимо добавить список сред для сборки. Если вы хотите собрать приложение для Mac OS X и 64-битной версии Windows 10, добавляемый фрагмент должен выглядеть так:


```
"runtimes": {  
  "win10-x64": {},  
  "osx.10.11-x64": {}  
}
```

Полный список идентификаторов пакетов находится по адресу <https://docs.microsoft.com/en-us/dotnet/articles/core/rid-catalog>.

При сборке или публикации приложения среду можно указать в командной строке при помощи флага `-r`:

```
dotnet publish -r win10-x64
```

После того как пакеты будут созданы, вы сможете сконцентрироваться на их развертывании в месте, в котором другие люди смогут пользоваться ими.

Несколько слов в пользу Azure

Существуют бесчисленные способы размещения приложений, чтобы с ними могли работать пользователи. В течение долгого времени люди и компании открывали собственные центры обработки данных. В таком подходе нет ничего плохого; собственно, многие люди продолжают его использовать. Проблема в том, чтобы найти подходящих людей, которые спроектируют такой центр и будут заниматься его сопровождением. При этом возникают некоторые физические проблемы — например, как правильно организовать подачу переменного тока высокого напряжения и насколько большой источник бесперебойного питания при этом понадобится. Кроме того, приходится решать ряд инфраструктурных проблем: как гарантировать наличие достаточного объема дискового пространства и как управлять пулом виртуальных машин. За управление резервным копированием часто отвечают сразу несколько людей, и у кого действительно есть время для тестирования процедуры восстановления данных?

Решения всех этих проблем не просты и не дешевы. Кроме того, эти проблемы не входят в основные направления деятельности компании. Если компания специализируется на добыче и очистке нефти, ее отдел информационных технологий (ИТ) все равно рассматривается как статья издержек, а не как неотъемлемая часть бизнеса, каким бы важным он ни был для успеха. Часто это заставляет компании искать решения для хостинга приложений за пределами внутренней ИТ-структуры.

В области внешнего хостинга существует целый спектр вариантов. На простейшем уровне компания может просто арендовать место для своих машин. Центр обработки данных обеспечивает физическую безопасность и решает некоторые инфраструктурные проблемы (такие, как охлаждение и питание), но почти ничего более. Часто в таких комплексах каждой компании предоставляется отдельное помещение, чтобы машины разных компаний были изолированы друг от друга.

Далее в спектре решений находятся компании, арендующие виртуальные машины или доступ к физическим машинам с ежемесячной оплатой. Они предоставляют дополнительные средства управления оборудованием, но вся ответственность за работу ПО, обеспечение достаточного пространства и вычислительных мощностей возлагается в основном на клиента.

Еще дальше в спектре следуют компании, предоставляющие услуги хостинга — по сути абстракцию над оборудованием, на котором работает ПО. В этом сценарии обычно арендуется не компьютер для сервера базы данных, а экземпляр базы данных с некоторыми гарантиями производительности.

Как аренда виртуальных машин, так и использование размещенных сервисов относятся к категории облачных технологий. Термин «облачные технологии» стал чрезвычайно перегруженным из-за большого количества маркетинговой путаницы. На самом деле он просто означает, что используемые вычислительные ресурсы виртуализированы и могут легко масштабироваться вверх или вниз. Оплата взимается только за фактически используемые ресурсы (а не за базовый уровень ресурсов, как в обычных центрах обработки данных). Такая схема организации вычислений называется *эластичными вычислениями* (elastic computing).

Эластичные вычисления идеально подходят для компаний с «пиковыми» нагрузками, то есть с чередующимися периодами интенсивного использования и минимальной нагрузки. Alpine Ski House — идеальный пример такой компании. В летние месяцы количество пользователей на сайте минимально из-за отсутствия снега, но с первыми заморозками спрос стремительно растет. При традиционном размещении Alpine Ski House пришлось бы платить за мощное оборудование весь год, хотя реально оно используется только часть года. Кроме того, нагрузка на сайт продажи абонементов намного выше утром, когда люди приезжают на склоны, а не днем, когда они уже катаются на склонах. Таким образом, интенсивность обращений к проекту Parsley изменяется не только в течение года; пики наблюдаются в течение одного дня. Без эластичного размещения Alpine Ski House пришлось бы подбирать мощность оборудования в соответствии с пиковой нагрузкой.

Платформа Microsoft Azure постоянно оказывается среди лидеров в отчетах Magic Quadrant Reports компании Gartner. Ее предложения IaaS (Infrastructure as a Service, «инфраструктура как сервис») находятся приблизительно на одном уровне с предложениями других поставщиков облачных сервисов, но по-настоящему выделяются предложения PaaS (Platform as a Service, «платформа как сервис»). IaaS — версия облачного размещения с наибольшими возможностями настройки, которая позволяет вам запускать собственные виртуальные машины и связывать их в специализированные сети, включающие выделенные каналы между Azure и локальными центрами обработки данных. На этой базе строятся предложения PaaS, предоставляющие абстрактную прослойку над IaaS. Эта прослойка включает такие аспекты, как масштабируемая база данных документов, служебная шина, сервисы, относящиеся к обработке аудио- и видеоданных, машинное обучение и многое другое.

Заимствуя превосходную аналогию у Скотта Ханселмана (Scott Hanselman), иметь собственный центр обработки данных — все равно что иметь собственный автомобиль: вы отвечаете за его заправку, техническое обслуживание и вождение. IaaS можно сравнить с прокатом машины. Вы по-прежнему отвечаете за вождение и заправку, но большая часть забот о техническом обслуживании с вас снимается. Наконец, PaaS напоминает поездку на такси. Вы просто сообщаете водителю, куда нужно ехать, а все остальное делается за вас. С таким сервисом, как DocumentDB, нет необходимости беспокоиться о масштабировании или резервном копировании; просто используйте уже имеющиеся средства.

Azure — чрезвычайно широкое семейство технологий с более чем 40 значительными продуктами. Мы сосредоточимся на одном из них: App Services.

Развертывание в Azure

Методов развертывания веб-приложений в Azure App Services достаточно. Файлы можно вручную копировать на сайт посредством FTP в стиле 1990-х годов, копировать в учетную запись Dropbox либо OneDrive, или же Azure может напрямую загружать приложение из репозитория с исходным кодом — такого, как Git (рис. 6.7).

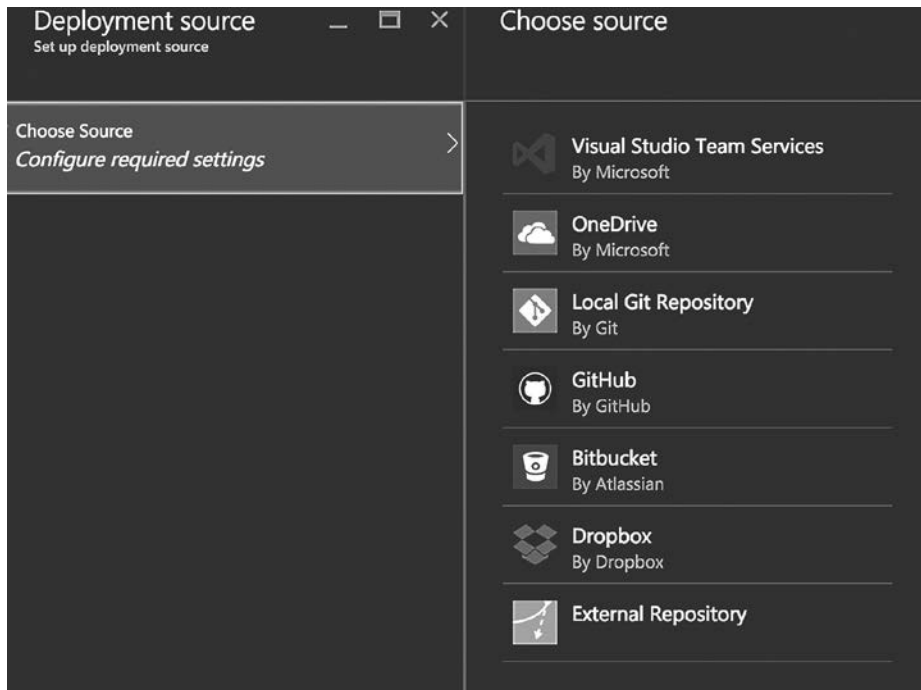


Рис. 6.7. Варианты источника развертывания в Microsoft Azure App Services

Применять OneDrive или Dropbox для развертывания всегда считалось дилетантством. Конечно, в некоторых ситуациях простое размещение файла в Dropbox — все, что необходимо для внесения изменения на «живом» веб-сайте, но есть и ситуации, когда клиент не разбирается в тонкостях, а просто ищет самый простой способ обновления сайта. Системы управления исходным кодом являются важной частью любой стратегии развертывания, поэтому возникает естественное желание использовать их для этих целей.

При настройке Azure App Services для публикации из системы управления исходным кодом простого сохранения в ветви системы достаточно для инициирования процессов сборки и развертывания. Вы можете настроить свою систему управления исходным кодом, создать в ней тестовую ветвь и основную ветвь, чтобы сохранение в тестовой ветви инициировало сборку с развертыванием на сервере тестирования, а сохранение в основной ветви инициировало сборку с развертыванием на «живом» сайте.

При этом система управления исходным кодом тесно связывается со сборкой и развертыванием. Становится очень просто случайно развернуть сайт, выполнив сохранение не в той ветви, причем нет гарантий, что содержимое тестовой ветви должно развертываться в основной ветви, когда настанет подходящий момент. В основной ветви могут присутствовать лишние сохранения, которые не были корректно адаптированы для тестирования, и это приведет к неожиданному поведению. В идеале двоичные файлы, развертываемые в продуктивной среде, точно совпадают с теми, которые развертываются в тестовой среде. Это предоставляет некоторую уверенность в том, что код продолжает функционировать в продуктивной среде так же, как и в тестовой.

Нам хотелось бы развернуть пакет для тестирования, провести с ним регрессионные тесты, а когда результат нас устроит — развернуть те же файлы в продуктивной среде. Существует много инструментов для решения этой задачи — таких, как TeamCity, Jenkins и Team Foundation Server. Для наших целей мы подключим свою учетную запись Azure к размещаемой версии TFS, которая теперь называется Visual Studio Team Services (VSTS).

В течение долгого времени средства сборки в Team Foundation Services были крайне несовершенными. Сборка определялась в файлах Workflow, которые было неудобно редактировать, а для отклонения от стандартной сборки требовалась высокая квалификация. Впрочем, с последней версией TFS управление сборкой стало в большей степени соответствовать подходу к сборке в Team City или Jenkins. Описания проще изменять, а библиотека готовых задач постоянно растет. Для многих сборок не требуется ничего, кроме объединения нескольких готовых фаз.

У нас уже имеется сборка, которая создает пакеты; остается лишь настроить хостинг для двоичных файлов.

ПРИМЕЧАНИЕ

Если у вас возникнут трудности с автоматическим подключением учетной записи Azure к VSTS, возможно, вам поможет статья по адресу <https://blogs.msdn.microsoft.com/visualstudioalm/2015/10/04/automating-azure-resource-group-deployment-using-a-service-principal-in-visual-studio-online-buildrelease-management/>.

Настройка VSTS для развертывания в Azure — относительно несложный процесс. В более сложных ситуациях развертывания, возможно, придется настроить несколько задач. А в нашем простом сценарии можно обойтись всего одной задачей в развертывании (рис. 6.8).

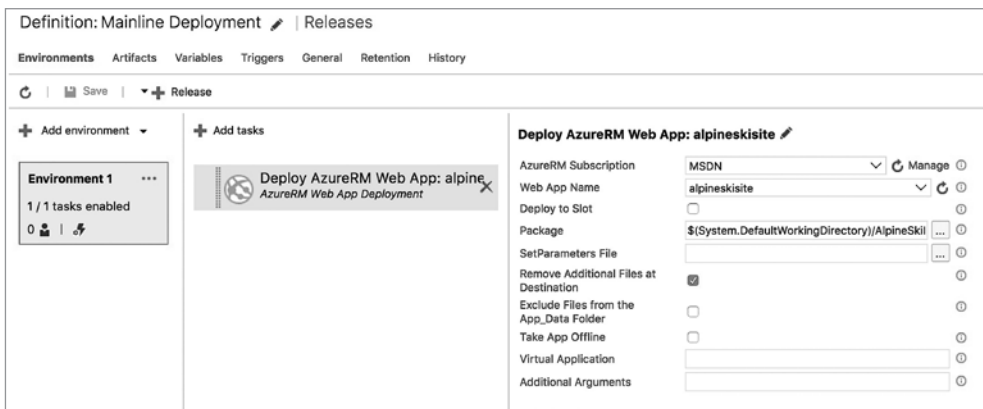


Рис. 6.8. Развертывание пакета в Microsoft Azure

С новым менеджером ресурсов достаточно указать имя веб-приложения и выбрать пакет для развертывания. Флажок **Remove Additional Files At Destination** (Удалить дополнительные файлы в целевом каталоге) всегда рекомендуется устанавливать — он гарантирует, что сборки всегда будут повторяемыми. Конечно, в этой простой конфигурации не хватает большого количества функциональности. Сборка требует, чтобы веб-приложение уже было создано, а следовательно, в процессе сборки понадобится ручное вмешательство. В идеале сборка создает ресурсы в процессе своей обработки. Это позволяет задать описание всей среды (веб-сервер, база данных, брандмауэры и т. д.) для целей тестирования в файле; таким образом вы получите полное, повторяемое развертывание вместо многочисленных щелчков при ручном управлении через портал.

Данная цель легко достигается при помощи шаблонов ARM (Azure Resource Manager). Они представляют собой файлы в формате JSON, определяющие целевые среды. Например, файл может содержать описание простого развертывания Umbraco, системы управления контентом (CMS) на базе ASP.NET. Вся совокуп-

ность виртуальных машин, сетевых инфраструктур и учетных записей определяется в 500-строчном документе JSON, который легко обрабатывается системами управления исходным кодом и версиями. Это пример описания инфраструктуры в коде. Ресурсы Azure объединяются в «группы ресурсов». Такие коллекции ресурсов позволяют выполнять логическую группировку всех ресурсов для заданной среды. Развертывание группы ресурсов из цикла выпуска может осуществляться сценарием PowerShell, или в случае использования VSTS — встроенной задачей. На рис. 6.9 представлены ресурсы, описанные шаблоном ARM.

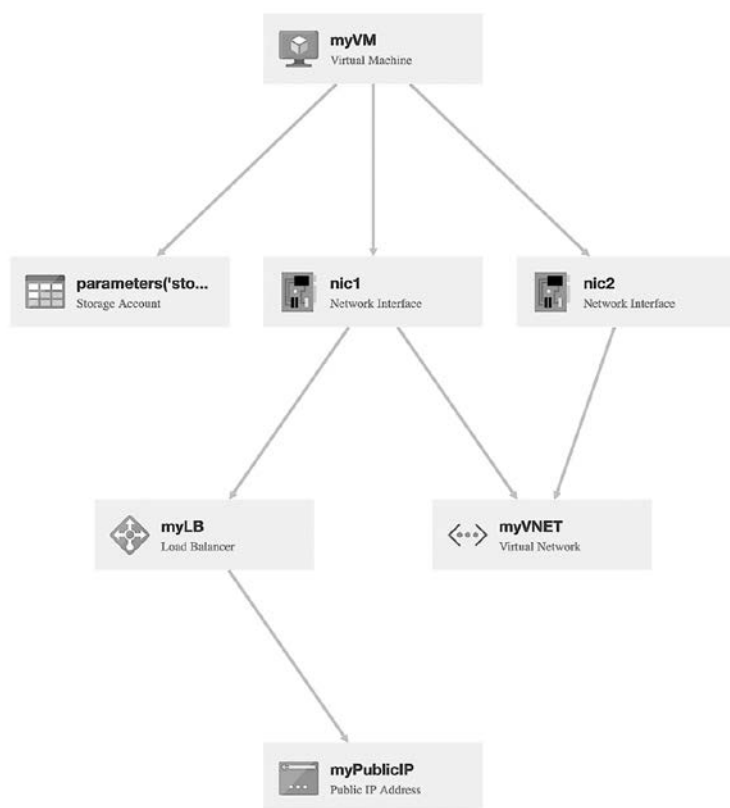


Рис. 6.9. Наглядное представление ресурсов, описанных шаблоном ARM

Шаблоны ARM также могут легко развертываться из сценариев PowerShell при наличии необходимого инструментария PowerShell для Azure. В следующей команде шаблон развертывается с административной учетной записью *skiadmin*.

```
New-AzureRmResourceGroupDeployment -ResourceGroupName alpineski -administratorLogin  
skiadmin -TemplateFile alpineski.json
```

Шаблоны могут быть параметризованными, что позволяет реализовать в них слегка различающиеся ситуации. Например, они могут получать количество создаваемых экземпляров веб-сервера: 1 для тестирования или 20 для продуктивной среды.

Существует бесчисленное множество других задач, которые могут добавляться в конвейер сборки. Например, у вас может возникнуть потребность в сине-зеленом развертывании (blue/green deployment), когда ваше приложение развертывается в горячем резерве перед переключением на него пользователей. А может быть, в процессе развертывания должны проводиться какие-то веб-тесты, которые подтвердят, что все действительно работает; в VSTS такая возможность тоже существует (рис. 6.10).

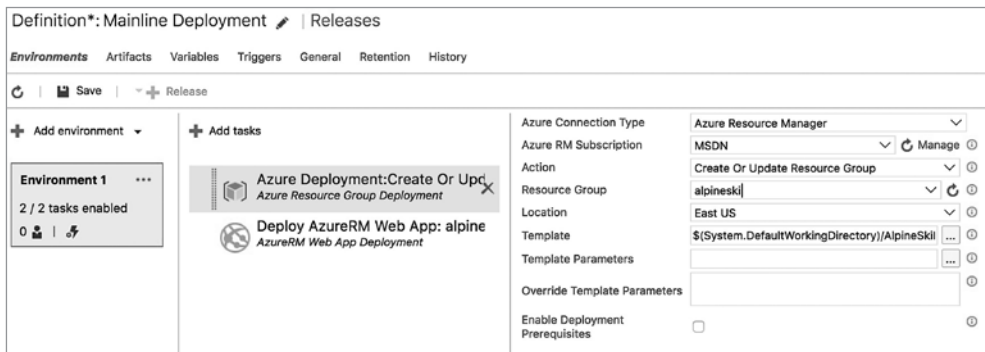


Рис. 6.10. Развертывание шаблона менеджера ресурсов из VSTS

Контейнерное развертывание

Если вы или ваша компания еще не готовы перейти на облачные технологии, существуют альтернативы, которые предоставляют некоторые преимущества облачных технологий в локальном варианте. Пожалуй, Docker и контейнеризация стали самыми впечатляющими изменениями в области развертывания за последнее десятилетие. Вся глава 9 «Контейнеры» посвящена использованию контейнеров для развертывания приложений.

Итоги

Публикация программных продуктов часто считается очень сложным делом. Некоторые компании с трудом выдают даже один выпуск в год из-за сложностей процесса. Движение непрерывного развертывания применяет к процессу выпуска старую поговорку «Если что-то идет плохо, делай это почаще». Сокращая вре-

мя между выпусками, мы заставляем себя оптимизировать и совершенствовать процесс, обычно за счет применения автоматизации. У быстрого развертывания много преимуществ: быстрое устранение дефектов безопасности, возможность тратить время на совершенствование продукта вместо возни с развертыванием, повторяемость выпусков за счет снижения вероятности ошибок при ручном развертывании.

И хотя эта глава в значительной мере посвящена развертыванию в Azure с использованием VSTS, здесь нет ничего такого, что нельзя было бы сделать средствами командной строки или других инструментов сборки. Важнейший вывод заключается в том, что сборка и развертывание могут и должны автоматизироваться. С ASP.NET Core процесс становится еще проще, потому что развертывание может осуществляться в более широком спектре платформ и без проблем с нарушением работы установленных приложений, использующих другие фреймворки.

ЧАСТЬ II

Спринт второй: Путешествие длиной в 1000 шагов

Глава 7. Сборка веб-приложений на платформе Microsoft Azure.....	112
Глава 8. Кроссплатформенность	130
Глава 9. Контейнеры.....	144
Глава 10. Entity Framework Core.....	160
Глава 11. Представления Razor.....	192
Глава 12. Конфигурация и журналирование	219

Даниэль была погружена в раздумья. Сейчас она дошла до середины сборника дисков сериала «Секретные материалы», который был куплен на распродаже в прошлом ноябре. Она только что посмотрела серию, в которой из-за утечки радиоактивных отходов на русском корабле появился гибрид человека и червя. Он нападал на всех, кто пользовался уборной, и утаскивал их... ей не хотелось думать, куда именно. Уж конечно, не в Нарнию. Она никогда не доверяла уборным с дырами в полу, так что это стало для нее последней каплей. Хватит с нее этих дыр, отныне только нормальная канализация... и возможно, стоит купить лекарство от червей-паразитов и держать его под рукой — просто на всякий случай.

К реальности ее вернуло чье-то покашливание. Сейчас должна была начаться ретроспектива первого спринта. Проект шел уже неделю, и что это была за неделя! Трудно было поверить, что всего неделю назад до нее дошли новости о новом проекте Parsley. Даниэль нравилось кодовое название, которое они придумали¹. Оно ассоциировалось со свежим новым началом; чем-то, что стирает вкус предыдущего блюда. Многие люди оставляют на тарелках петрушку, которую им кладут в модных ресторанах, но Даниэль всегда ее съедала.

¹ Parsley — петрушка (травя). — *Примеч. пер.*

Балаш стоял у доски в центре комнаты. Беседа с Тимом, он помахивал в воздухе маркером, словно приносиваясь. Из-за небольшой ошибки в заказе, допущенной почти 10 лет назад, курорт приобрел огромное количество маркеров с убийным запахом чернил (в те времена это было нормой). Впрочем, писали маркеры хорошо.

«Ладно, начинаем, — призвал Балаш. — Цель ретроспективы — разобраться в том, что мы должны добавить в свою работу, исключить из нее или оставить без изменений». Он разделил доску на три столбца и подписал их «Начать», «Прекратить» и «Продолжать».

«А теперь все берут маркеры и начинают писать! Кстати, отличные маркеры».

Утвердившись в своих подозрениях, Даниэль взяла зеленый маркер и направилась к доске. В графе «Продолжать» она написала «Использовать .NET» и «Проводить непрерывное развертывание». Другие участники команды тоже брали маркеры, пытались оттеснить конкурентов и подобраться к доске. Вот что у них получилось через несколько минут:

Начать	Прекратить	Продолжать
Получить реальные результаты	Утренние собрания... разве нельзя их проводить по вечерам?	Использовать .NET
Выращивать салат в комнате отдыха	Работать только над тем, чтобы добиться успешного построения	Проводить непрерывное развертывание
Проводить лекции по принципам MVC и других паттернов (во время обеда?)	Возиться с Linux (слишком сложно)	Использовать Azure
Чаще применять паттерны передачи сообщений	Использовать Windows (не хватает мощности)	

Многие предложения были шуточными, но это также было признаком того, что в команде установилась непринужденная атмосфера.

Видимо, Балаш был доволен результатом. «Итак, у нас нет по-настоящему серьезных проблем, над которыми нужно работать. Есть тема, за которую нужно браться, но я думаю, мы справляемся намного лучше, чем все считают. Нам удалось построить надежный конвейер построения, который в будущем устранил все препятствия в работе над кодом».

Он продолжал: «Я видел, как многие команды не занимались развертыванием на ранней стадии процесса. Потом они в спешке последних минут выдавали код, который работал локально и развертывался в рабочей среде».

На Марка-1 это не произвело особого впечатления: «Пока что нам удалось развернуть сайт, который не делает ничего. Не знаю, как мы собираемся выдерживать сроки с таким подходом».

«Знаю, как это выглядит, — сказал Балаш, — но написание кода — это как раз то, чему будет посвящен наш второй спринт».

«Да уж, — сказал Марк-1. — Жду не дождусь, когда начнется нормальная работа».

«Прекрасно! — ответил Балаш. — Эта энергия нам пригодится на следующем спринте. Вижу, также есть потребность в повышении квалификации для ознакомления команды с некоторыми концепциями, лежащими в основе MVC. Я думаю, у нас в команде есть очень сильные разработчики, которые уже работали в этой области. С этим можно что-нибудь придумать. Уверен, на обеды мы тоже можем раскошелиться». Балаш переглянулся с Тимом, который кивнул в знак согласия. Тим был не из тех, кто отказывается от бесплатных обедов.

«Я рад, что платформа Azure так популярна, — продолжал Балаш, — потому что руководство довольно ее схемой ценообразования. Я знаю, Тим хочет убрать все оборудование из серверной и переместить в облако».

«Да, — сказал Тим. — Нам трудно управлять серверами здесь — комплектующих приходится ждать слишком долго, а электричество стоит сравнительно дорого. Вдобавок теперь, когда в штате появилось столько новых людей, нам понадобится офисное пространство, поэтому освобождение серверной принесет большую пользу».

«Хорошо, — сказал Балаш. — Почти в каждом направлении понятно, куда нужно двигаться. За работу — пора писать код!»

«С салатом в комнате отдыха мы пока не разобрались, — пожаловалась Даниэль, — но кажется, проект идет нормально».

7

Сборка веб-приложений на платформе Microsoft Azure

«Отличная работа, Даниэль. Что-то понемногу прорисовывается. — Марк-1 просматривал описания этапов сборки, сконфигурированных Даниэль для упрощения процесса. — Сколько времени уходит на выполнение?»

«От слияния до развертывания? Около четырех минут», — Даниэль почувствовала, что ей полагается приз «За упрощение работы» — и по праву.

«Вот как, значит? — задумчиво продолжал Марк-1. Он знал, что в следующем спринте им предстоит работа по передаче данных карт стороннему подрядчику, который должен печатать карты, и решил на всякий случай разведать обстановку. — Мы перешли на облачные технологии. И... это круто, наверное. Я хочу сказать, что нам придется разобраться со многими проблемами, но результат должен быть крутым. Надеюсь. Верно? — Он сделал неловкую паузу, а потом продолжал с некоторым скептицизмом. — Так где мы будем хранить экспортируемые данные карт? Ведь файловой системы у нас не будет?»

«Не будет, ты прав. В нашем развертывании есть файловая система, но дисковое пространство не бесконечно. Думаю, у нас есть несколько вариантов. В принципе данные можно поставить в очереди и обрабатывать по мере надобности, но мне этот вариант кажется сомнительным, — сказала Даниэль. — Может, есть смысл опробовать табличное хранилище? Или хранилище документов? Не хочу заставлять всех выбирать технологию только для того, чтобы опробовать что-то новое».

«Точно! — сказал Марк-1, довольный тем, что у него нашлись единомышленники. — В последнее время здесь вообще летает много новомодных терминов. — Внезапно он заговорил намного тише: — Ты не поверишь, ко мне подошел Тим и предложил присмотреться к службе Event! Он говорил о том, что клиентов нужно рассматривать как потоки событий в "Интернете вещей". Не уверен, что он понимает, чем мы тут занимаемся. Но ведь по сути данные карт должны передаваться в формате CSV, верно? Так почему мы не записываем файлы CSV?»

«Да, я понимаю, о чем ты. Конечно, архитектура космического уровня тут не нужна. Что ты думаешь о BLOB-хранилище? Из того, что я видела, использовать его немногим сложнее, чем настроить базу данных».

«Пожалуй, в этом что-то есть, — ответил он. — Наши выходные данные — файлы. Будем относиться к ним как к файлам и сохранять их в Azure в месте для файлов. Мне кажется, что это самое разумное».

Уверенная в правильности выбранного направления, Даниэль перекатила свое кресло к рабочему месту Марка-1. Они создали новую ветвь и взялись за работу.

Что такое PaaS

Если вы впервые соприкоснулись с предложениями поставщиков облачных сервисов или возвращаетесь к облачным технологиям через несколько лет работы в других областях, вам покажется, что здесь появилось много красивых блестящих штучек, которые так и хочется опробовать. И это действительно так, но чрезмерное увлечение новыми технологиями подвело очень многие приложения.

Есть хорошее правило, которое поможет вам оценить эти замечательные новые возможности, — используйте то, что вы сможете без проблем спроектировать. Это не означает, что вы должны постоянно держаться в своей «зоне комфорта». Более того, мы рекомендуем выходить за пределы того, что казалось вам комфортным в последнем проекте. По сути это правило означает, что выбор используемых возможностей должен соответствовать вашему проекту. Обычно мы обнаруживали, что в проекте лучше всего задействовать те возможности, которые заполняют конкретную потребность и при этом оказываются как раз впору (в отличие от тех, которые нуждаются в подгонке).

По мере становления вашего приложения часто проявляются закономерности и сервисы, которые естественным образом формируются в зависимости от типа выполняемой работы. Например, при выполнении пакетной обработки вы упрощаете создание пакетов. Даже сама среда .NET Framework встраивает эти виды сервисов в библиотеку базовых классов, поэтому-то нам больше не приходится писать библиотеки для клиентов FTP.

Именно на такие паттерны и сервисы следует обращать внимание в облачных предложениях. Не пытайтесь затыкать квадратные отверстия круглыми пробками — подобно тому, как Даниэль и Марк-1 стараются не рассматривать клиентов горнолыжного курорта как счетчики воды, с которых снимаются показания. На платформе Azure предложения PaaS (Platform as a Service, «платформа как сервис») обычно позволяют решать задачи многих размеров и классов без необходимости «изобретать велосипед».

Платформенные сервисы

В организациях иногда применяется стратегия «поднять и перенести»; это означает, что организация берет все, что работает локально в ее среде, и переносит это в облако. В одних компаниях это означает развертывание на виртуальной машине

в облаке, в других — миграцию с физического оборудования на виртуальное, с последующим переносом виртуальной машины в облако. Но сразу проясним: если вы реализуете стратегию «поднять и перенести» и будете утверждать, что у вас есть облачная стратегия, — это не так. Иначе говоря, если вы всего лишь перенесли свои виртуальные машины на AWS или Azure, вы не используете ни структуру, ни программные предложения, оптимизированные для облака. Стратегия «поднять и перенести» — небольшая часть истории, которая принципиально отличается от виртуализации внутри компании. Правда, она избавляет от затрат на покупку нового серверного оборудования и поэтому может показаться хорошим вариантом с экономической точки зрения, но она не использует многие преимущества от перехода на облачные технологии.

Как бы сурово ни прозвучал последний абзац, стратегия «поднять и перенести» может стать хорошей частью долгосрочного плана перемещения в облако. Например, после того как ваше приложение будет развернуто на виртуальной машине в Microsoft Azure, вы можете начать пользоваться очередями для доставки сообщений новым сервисам, которые вы сами построите, или же записывать файлы журналов в BLOB-хранилище. Со временем вы переместите свой сайт в веб-приложение или упакуете свой API в сервисы API Management, чтобы им могли пользоваться другие организации. Постепенно ваше приложение начнет в меньшей степени зависеть от инфраструктуры (виртуальных машин, на которых проведено развертывание) и будет двигаться в направлении выполнения на платформенных сервисах, предоставляемых облаком.

Откровенно говоря, перечисление всех сервисов, доступных в облаке, напоминает попытку посчитать муравьев в колонии. Их численность меняется, они постоянно двигаются, а иногда порождают совершенно новое поколение сервисов, о которых вы можете не знать. Тем не менее в табл. 7.1 перечислены некоторые интересные сервисы, проверенные временем, которые вы можете начать встраивать в свое приложение.

Выбрав принцип компоновки из блоков, вы сможете быстро спроектировать приложение, и вам не придется заниматься утомительной реализацией всех типовых фрагментов, являющихся частью каждого проекта. В 1999 году Джеймс был нанят в команду из двух человек для проектирования простого веб-приложения CRUD (Create/Read/Update/Delete), которое предназначалось для упрощения сопровождения нескольких таблиц в базе данных. Сайт должен был защищаться пользовательскими удостоверениями и поддерживать две пользовательские роли, причем одной группе было разрешено добавление в таблицу категорий. Работа была едва завершена в отведенные для нее 120 часов. В наши дни для создания CRUD-приложения можно воспользоваться Microsoft Azure AD B2C для аутентификации, Microsoft Azure SQL Database для резервного копирования данных, с объединением трех контроллеров в приложении ASP.NET Core MVC. Вы може-

те сделать все это и создать свое приложение с тем же уровнем функциональности менее чем за 4 часа.

У Microsoft Azure есть более 100 аспектов, которыми вы можете пользоваться при проектировании своих приложений, но для успеха проекта очень важно объединить их способом, обеспечивающим повторяемость.

Таблица 7.1. Подборка облачных предложений в Microsoft Azure

Категория	Сервисы и предложения
Вычисления и хранение данных	Пакеты Service Fabric Функции Blob Очереди Хранилище «озеро данных»
Сети, мобильные и веб-технологии	CDN, Azure DNS Распределитель нагрузки App Service API Management Notification Hubs
Данные и корпоративная интеграция	SQL Database DocumentDB SQL Data Warehouse Кэш Redis Сервисная шина Logic Apps
Искусственный интеллект и анализ данных	Машинное обучение Cognitive Services (Vision, Emotion и Face API)
«Интернет вещей»	Stream Analytics Event Hubs
Безопасность, идентификация и мониторинг	Azure AD Azure AD B2C Multi-Factor Authentication Access Control Service Key Vault Резервное копирование AutoScale Azure Resource Manager
Инструменты разработчика	Visual Studio Team Services Нагрузочное тестирование Azure DevTest Labs HockeyApp

Оснастка, уничтожение и воссоздание сервисов

В главе 6 «Развертывание» мы рассмотрели шаблоны Microsoft Azure Resource Manager и возможности их применения в процессе развертывания. Шаблоны имеют формат JSON и содержат всю информацию, необходимую для создания нового экземпляра настроенного вами сервиса. Шаблоны параметризуются таким образом, чтобы вы могли использовать внешний файл JSON для адаптации ресурса к любой среде, в которой он будет развертываться. Но после идентификации ресурса, который должен быть включен в развертывание, как построить шаблон для использования в вашем сценарии?

К счастью, делать это вручную не придется. Любой сервис, созданный как часть группы ресурсов (Resource Group) в Microsoft Azure, можно просмотреть в мощной программе Azure Resource Explorer (<https://resources.azure.com>). Программа позволяет просмотреть любой ресурс, который вы создали (рис. 7.1), и извлечь нужный шаблон со всеми заранее настроенными параметрами и настройками.

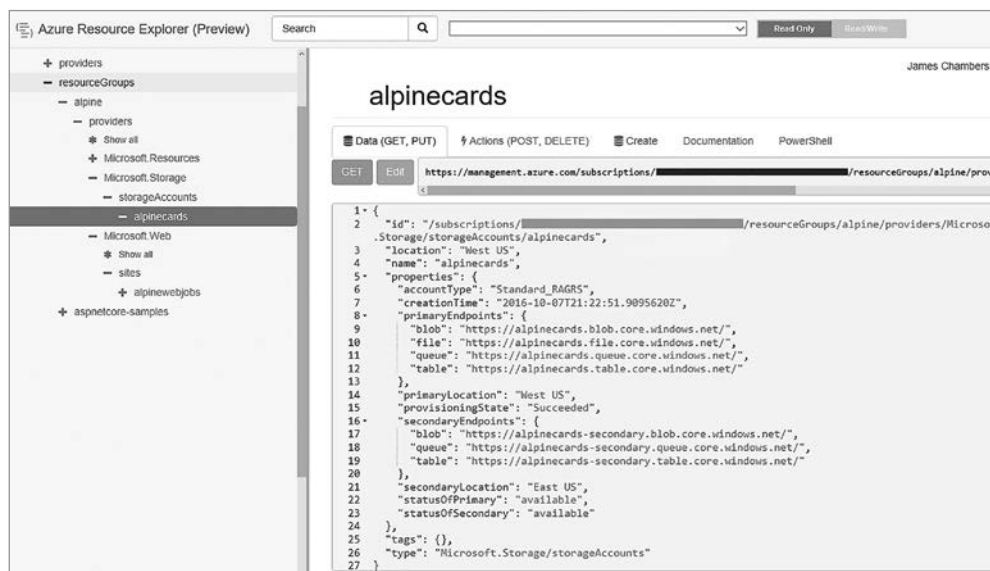


Рис. 7.1. Azure Resource Explorer

Возможен и другой вариант: чтобы начать работу с шаблоном, вам не обязательно проходить весь процесс создания ресурса до конца. На рис. 7.2 показано, как Azure Portal используется не только для генерирования шаблона, но и для всего исходного кода, необходимого в .NET, Ruby, PowerShell или CLI. Чтобы вызвать этот экран, выберите команду Automation Options перед завершением создания сервиса.

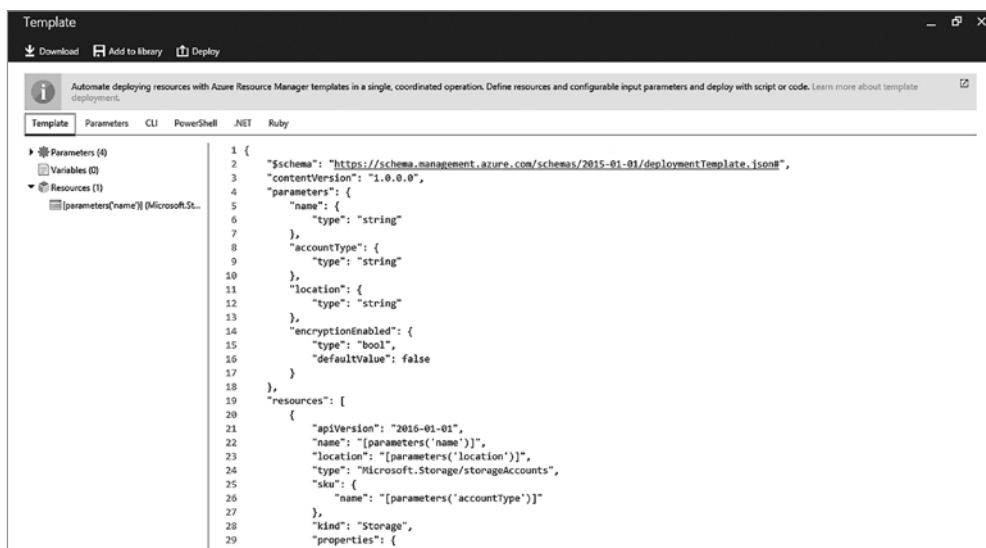


Рис. 7.2. Просмотр шаблона в Azure Portal

Используйте шаблоны и загружаемые файлы параметров для создания сервисов и обновления сценариев сборки по мере необходимости. За дополнительной информацией о создании и настройке шаблонов и параметров обращайтесь по адресу <https://azure.microsoft.com/en-us/documentation/articles/resource-group-authoring-templates/>.

Проектирование приложений с использованием Platform Services

Рассмотрим одну из функций приложения, которую необходимо реализовать команде Alpine Ski House. Когда пользователи создают карты для своих учетных записей, им предлагается отправить фотографию для профиля. Позднее карты печатаются сторонней компанией, которая использует процесс лазерной печати для нанесения черно-белой фотографии владельца на карту. Отправленная фотография масштабируется до определенных размеров и преобразуется в оттенки серого с использованием конкретного стандарта преобразования, обеспечивающего наилучшее качество печати.

Команда Alpine Ski House нашла для своих целей пару вспомогательных компонентов Microsoft Azure, включая простую учетную запись хранения (storage account) с обработкой, поддерживаемой App Services. Учетные записи хранения предоставляют возможность сохранения произвольных данных с использованием двоичных больших объектов, или BLOB (Binary Large Object). Сервис BLOB также предоставляет вспомогательное хранилище для WebJobs — части App Services,

предлагающей разные механизмы обработки планируемых или запускаемых по условию заданий.

Сервис BLOB в учетных записях хранения — всего лишь одна из возможностей. Учетные записи хранения также предоставляют доступ к неструктурированным данным в форме табличного хранилища NoSQL, файлового хранилища и, наконец, простого канала передачи данных через очереди.

WebJobs входит в набор компонентов App Service, который также включает мобильные сервисы, разработку API и возможность хостинга сайтов в масштабе, который предлагает Web Apps. App Service поддерживает не только языки .NET, но также Java, Node.js, Python и PHP. В App Service можно создать один или несколько компонентов, входящих в одно множество уровней производительности и масштабирования, и все сервисы, созданные вами в экземпляре, будут отнесены к одной группе ограничений на использование, исходящий трафик и выставление счетов.

Вооружившись BLOB и заданиями, команда Apline связала их триггером, основанным на сообщении в очереди. Чтобы вся эта схема заработала, мы воспользуемся несколькими компонентами, часть из которых более подробно рассматривается позже в книге. После того как вы освоите эти компоненты, вы согласитесь, что подобные решения — композиции сервисов, предоставляемых экосистемой Azure, — бывают достаточно элегантными.

Создание учетной записи хранения

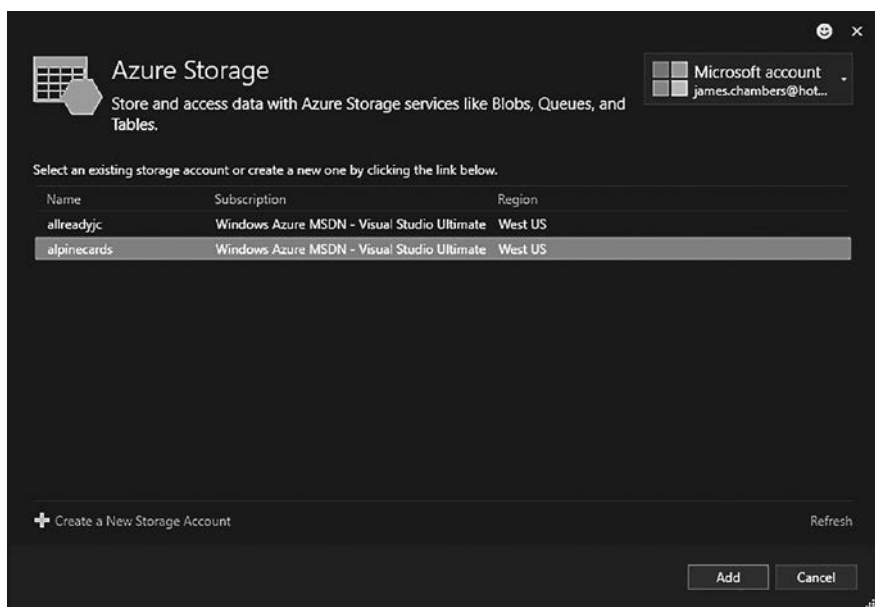
Чтобы начать пользоваться этими платформенными функциями, сначала необходимо создать учетную запись хранения при помощи Azure Portal, CLI, PowerShell или Visual Studio с диалоговым окном **Connected Services**.

Чтобы воспользоваться Visual Studio, щелкните правой кнопкой мыши на проекте в Solution Explorer и выберите команду **Add**, затем выберите **Connected Service**. На экране появится начальное окно программы-мастера, похожее на изображенное на рис. 7.3. Если вы уже вошли в свою учетную запись Microsoft и с этой учетной записью связана подписка Azure, у вас будет возможность воспользоваться существующей учетной записью хранения или же создать новую во время работы.

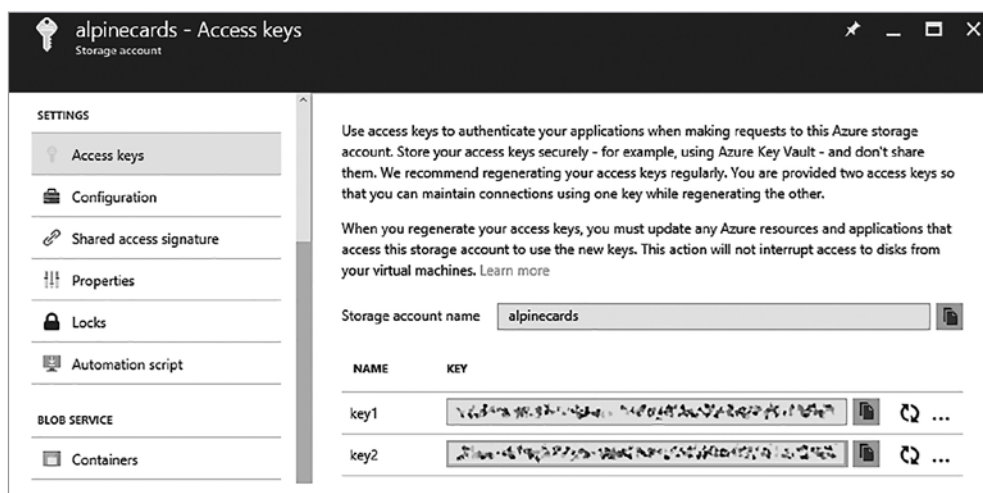
После завершения работы с мастером ваш проект дополняется необходимыми пакетами и файлом конфигурации **config.json**, который содержит строку подключения, необходимую вашему приложению для связи с учетной записью хранения из вашей подписки.

ПРИМЕЧАНИЕ

Переместите информацию строки подключения в своем проекте из места, в котором диалоговое окно **Connected Services** сохраняет ее по умолчанию. Автоматическое добавление информации удобно, но оставлять конфиденциальные сведения в файле, хранящемся в системе управления исходным кодом, не рекомендуется. Защита конфиденциальной информации более подробно рассматривается в главе 12 «Конфигурация и журналирование».

**Рис. 7.3.** Мастер Azure Storage Connected Service

Если вы не хотите использовать мастер Connected Services Wizard, прочитайте строку подключения для учетной записи хранения из Azure Portal. Строки подключения называются `key1` и `key2` и находятся в разделе `Access keys` группы параметров `Settings` (рис. 7.4). Строки подключения образуют часть конфигурации

**Рис. 7.4.** Обращение к ключам доступа для учетной записи хранения

проекта (эта тема рассматривается в главе 12) и предоставляются вашим сервисам через паттерн, называемый инверсией управления; он будет рассмотрен в главе 14 «Внедрение зависимостей». За примером механики конфигурации в действии обращайтесь к классу `AlpineConfigurationBuilder`, в котором настройки загружаются в конфигурационную коллекцию, а также к классу `BlobFileUploadService`, в котором строка непосредственно разбирается и используется приложением.

Также необходимо добавить в проект несколько пакетов из `project.json`; выберите команду **Manage NuGet Packages** в контекстном меню (открывается щелчком правой кнопки мыши на проекте в `Solution Explorer`) или в `Package Manager Console` в `Visual Studio`. Если у вас нет опыта работы с NuGet, откройте главу 14 «Внедрение зависимостей» или вернитесь к этой главе позже. А сейчас вам понадобятся следующие пакеты:

- ☐ `WindowsAzure.Storage`
- ☐ `System.Spatial`
- ☐ `Microsoft.Data.Edm`
- ☐ `Microsoft.Data.OData`
- ☐ `Microsoft.Data.Services.Client`

Ознакомившись с проектом, прилагаемым к книге, вы увидите, что команда также создала консольное приложение, которое развертывается как веб-задание для приложения. Приложение также получает зависимости от следующих пакетов:

- ☐ `Microsoft.Azure.WebJobs`
- ☐ `ImageProcessorCore`

Пакет `WebJobs` добавляет вспомогательные классы и атрибуты, предназначенные для ускорения создания задач автоматизации, развертываемых в `Azure`. `ImageProcessorCore` — ASP.NET Core-совместимая библиотека обработки графики, которая в настоящее время разрабатывается сообществом открытого кода и служит заменой для `System.Drawing`. Оба пакета будут рассмотрены позже в этой главе.

Хранение изображений в BLOB-контейнерах

Даниэль завершила работу над представлением формы для ввода данных карты и добавила элемент формы для отправки файла в процессе создания карты (листинг 7.1). В представлении используются тег-хелперы — возможность `Razor` и `MVC`, кратко представленная в главе 11 «Представления `Razor`» и подробно рассматриваемая в главе 19 «Многоразовый код».

Листинг 7.1. Обновление представления для сохранения отправленного изображения

```
// Новое поле в представлении, Views/SkiCard/Create.cshtml
<div class="form-group">
  <label asp-for="CardImage" class="col-md-2 control-label"></label>
  <div class="col-md-10">
    <input asp-for="CardImage" />
  </div>
</div>
```

Благодаря привязке модели контроллер не нуждается в обновлении, потому что тип элемента HTML `file` связан со свойством `IFormFile` нашей модели. В листинге 7.2 свойство `IFormFile` используется для отображения свойства в расширенном классе модели представления.

Листинг 7.2. Обновления в модели представления для поддержки автоматической привязки модели

```
// Новое поле как свойство модели представления, CreateSkiCardViewModel.cs
[Display(Name = "Image to be Displayed on Card")]
public IFormFile CardImage { set; get; }
```

Мы можем использовать `BlobFileUploadService` в этом контроллере для вызова одного метода `UploadFileFromStream`, который берет на себя весь процесс отправки файла. Фрагмент метода `UploadFileFromStream`, приведенный в листинге 7.3, создает подключение к контейнеру и осуществляет асинхронную отправку файла. Чтобы убедиться в существовании контейнера, мы используем `CreateIfNotExistsAsync` — облегченный вызов с минимальными непроизводительными затратами.

Листинг 7.3. Отправка изображения в BLOB-хранилище в Azure

```
// подготовка клиента
var storageAccount = CloudStorageAccount.Parse(_storageSettings.
AzureStorageConnectionString);
var blobClient = storageAccount.CreateCloudBlobClient();
var container = blobClient.GetContainerReference(containerName);
await container.CreateIfNotExistsAsync();

// сохранение изображения в контейнере
var blob = container.GetBlockBlobReference(targetFilename);
await blob.UploadFromStreamAsync(imageStream);
```

В целях оптимизации код, проверяющий существование облачных ресурсов, может быть перемещен в процедуру инициализации, выполняемую при запуске приложения. И хотя вызов `CreateIfNotExistsAsync` «на месте» не создает особых проблем, он также не является обязательным при каждом подключении к сервису.

Интеграция очередей хранилища

В конце вызова `BlobFileUploadService.UploadFileFromStream` инициируется событие, которое обрабатывается другим блоком кода в приложении. Подробности механизма сообщений сейчас не слишком важны, но обработчик, который реагирует на событие, заслуживает внимания. В данном случае приложение передает ответственность за обработку графики за пределы текущего запроса. Мы хотим, чтобы приложение реагировало на действия пользователя по возможности быстро и плавно, а для этого все операции, связанные с ресурсами, должны быть вынесены в другие процессы.

В листинге 7.4 приведена часть кода класса `QueueResizeOnSkiCardImageUploadedHandler`. По аналогии с кодом, который использовался для подключения к BLOB-сервисам, необходимо создать экземпляр клиента и убедиться в том, что ресурс, к которому мы пытаемся обратиться, действительно существует.

Листинг 7.4. Отправка сообщения в очередь сообщений Azure

```
// подготовка клиента очереди
var storageAccount = CloudStorageAccount.Parse(_storageSettings.
AzureStorageConnectionString);
var queueClient = storageAccount.CreateCloudQueueClient();
var imageQueue = queueClient.GetQueueReference("skicard-imageprocessing");
await imageQueue.CreateIfNotExistsAsync();

// подготовка и отправка сообщения в очередь
var message = new CloudQueueMessage(notification.FileName);
await imageQueue.AddMessageAsync(message);
```

После того как подключение будет создано, мы создаем сообщение с именем отправленного файла, после чего возвращаемся к основной работе веб-сайта — отображению страниц. Также отправляется сообщение, сигнализирующее, что работа должна быть выполнена другим сервисом, после чего приложение возвращается к визуализации представлений. Чтобы продолжить цепочку обработки изображения, необходимо перейти к проекту веб-задания, созданному для обработки изображений за пределами основной рабочей нагрузки веб-сервера.

Автоматизация обработки с использованием Azure WebJobs

Время от времени в нашей экосистеме встречается разработка, которая радикально изменяет подход к решению задач. Мы полагаем, что эволюция Azure WebJobs является как раз таким примером.

ПРИМЕЧАНИЕ

Технология WebJobs продолжает развиваться и в будущем будет называться Azure Functions. Технология Azure Functions будет предоставлять новые возможности, улучшенную изоляцию и функциональность за пределами Web Apps, но в настоящее время она находится в состоянии предварительного ознакомления и еще доступна не во всех регионах Azure. За дополнительной информацией об Azure Functions обращайтесь по адресу <https://azure.microsoft.com/en-us/services/functions/>.

Технология WebJobs появилась ввиду необходимости запуска произвольного кода, который нередко занимает много времени: обработки полученных файлов, разбора содержимого блогов или отправки сообщений электронной почты с определенной периодичностью (вместо установленного графика), постоянного выполнения в фоновом режиме или чего-то, что срабатывает по событию в приложении или извне. Веб-заданием может стать любой исполняемый файл или файл сценария, созданный вами. Код, который должен выполняться, упаковывается в ZIP-файл и отправляется Azure.

Для Даниэль и Марка-1 именно здесь стал проявляться потенциал концепции PaaS. Им не нужно было писать служебный код взаимодействия с очередями или чтения BLOB-объектов из контейнера, поэтому обработка очередей изображений выполнялась всего в трех строках кода (листинг 7.5). После извлечения WebJobs SDK из пакета NuGet, добавленного ранее, они смогли сжать то, что обычно представляет собой более 40 строк кода конфигурации, подключения и обработки, до следующего короткого фрагмента.

Листинг 7.5. Веб-задание Azure для обработки заданий, инициируемое через очередь

```
public class ImageAdjustments
{
    public static void ResizeAndGrayscaleImage(
        [QueueTrigger("skicard-imageprocessing")] string message,
        [Blob("cardimages/{queueTrigger}", FileAccess.Read)] Stream imageStream,
        [Blob("processed/{queueTrigger}", FileAccess.Write)] Stream
        resizedImageStream)
    {
        var image = new Image(imageStream);

        var resizedImage = image
            .Resize(300, 0)
            .Grayscale(GrayscaleMode.Bt709);

        resizedImage.SaveAsJpeg(resizedImageStream);
    }
}
```

Конечно, на бумаге получается больше трех строк, но с точки зрения компилятора этот блок кода состоит из двух созданий переменных и вызова `SaveAsJpeg`. Параметры в сигнатуре метода помечаются атрибутами из WebJobs SDK:

- `QueueTriggerAttribute`: показывает, какой тип события, относящегося к хранилищу, инициирует сигнал события и выполняет код. Атрибуту передается имя очереди для прослушивания.
- `BlobAttribute`: используется для заполнения параметра из расположения, заданного в шаблоне, в данном случае с разрешениями чтения/записи для потоков. SDK берет на себя создание и наполнение потоков, а также уборку «мусора».

Также стоит заметить, что эти три строки кода стали возможны благодаря растущей популярности библиотеки `ImageProcessing`, разработанной Джеймсом Джексон-Саутом (James Jackson-South), и живому, растущему списку участников из сообщества открытого кода по адресу <https://github.com/JimBobSquarePants/ImageProcessor>. Изображение инициализируется по входному потоку и сохраняется в переменной `image`. Затем мы используем синтаксис библиотеки, чтобы сначала изменить размеры изображения до ширины 300 пикселей с сохранением пропорций (параметр высоты равен 0), после чего преобразуем в цветовое пространство оттенков серого. Наконец, изображение сохраняется в выходном потоке.

Для объяснения всего происходящего требуется гораздо больше слов, чем строк кода, но в итоге с минимумом усилий и использованием нескольких платформенных сервисов нам удалось создать быстрое веб-приложение, способное передать основную часть обработки внешней функциональности.

Масштабирование приложений

Как упоминалось в предыдущей главе, эластичные облачные технологии — то, что позволяет небольшим организациям с периодами резкого возрастания трафика приспособиться к перепадам трафика и переменным потребностям в обработке. Для организаций среднего размера — таких, как `Alpine Ski House`, — эластичность платформы обеспечивает возможность роста и расширения предлагаемых услуг без необходимости заранее оплачивать инфраструктуру, которая их поддерживает. Простое расширение и сокращение по размерам нагрузки также способно сэкономить крупным организациям огромные суммы, потому что им приходится платить только за реально необходимые вычислительные мощности, без оплаты за время пониженной нагрузки или прямые простои, когда бизнес-процессы не активны.

Масштабирование в нескольких направлениях

Один из самых полезных аспектов использования PaaS-подхода к разработке — простота масштабирования ресурсов. Хотя поддерживать работу виртуальных машин и изменять их размеры через Azure Resource Manager становится проще, во многих случаях существуют ограничения, связанные с размерами и кластерами развертывания; с классическими образами виртуальных машин таких ограничений еще больше. Масштабируемые наборы виртуальных машин предоставляют организациям возможность запускать виртуальные машины «на лету» в соответствии с потребностями, но при этом вы в основном продолжаете отвечать за сопровождение операционной системы и другие аспекты своего парка виртуальных серверов. PaaS позволяет строить решение с других позиций, предоставляя набор сервисов, готовых к использованию, который может масштабироваться по вертикали или горизонтали, не беспокоясь об оборудовании или операционной системе, на которых они базируются.

Вертикальное масштабирование напоминает покупку более мощного оборудования для фермы серверов. С точки зрения инфраструктуры в пространстве виртуальных машин существуют разные ценовые уровни — от нескольких десятков долларов в месяц до нескольких тысяч долларов в месяц за каждую машину. Выбор уровня и размера оплаты также действует в пространстве PaaS, где вы можете выбирать из бесплатных услуг на сервисе начального уровня или же переходить на платные уровни с базовыми (Basic), стандартными (Standard) и премиальными (Premium) возможностями оборудования. В обоих сценариях действует принцип: чем больше денег вы тратите, тем больше в вашем распоряжении будет процессоров, вычислительных мощностей и памяти. Разные уровни также открывают доступ к расширению входного и выходного трафика, дискового пространства и дополнительным функциям: безопасности, сервису интеграции и личным доменам.

С другой стороны, горизонтальное масштабирование можно сравнить с приобретением большего количества серверов для вашей фермы серверов. Вместо того чтобы тратить деньги на повышение мощности одной машины, вы тратите деньги на покупку дополнительных машин, которые помогают распределить рабочую нагрузку в надежде на то, что она будет равномерно распределяться и обеспечит параллельную обработку. В Azure машины и сервисы, которые задействованы в масштабировании, имеют такой же размер, как первая, поэтому оценка общих затрат при горизонтальном масштабировании может вычисляться как величина, кратная размеру машины.

Если вы собираетесь экспериментировать с разными вариантами ценообразования и семействами, вы можете воспользоваться калькулятором ценообразования Microsoft Azure по адресу <https://azure.microsoft.com/en-us/pricing/calculator/>.

Эластичное масштабирование

После выбора вертикального уровня масштабирования (или вычислительной мощности) вы можете объединить свое решение с горизонтальным масштабированием (количеством машин) для достижения эластичного масштабирования. Как нетрудно догадаться, большие затраты дают больше возможностей; на рис. 7.5 приведены примеры возможностей на премиальном, стандартном и базовом тарифах. Для масштабирования необходим платный тариф, а для поддержки автоматического масштабирования — как минимум стандартный тариф.

P1 Premium		S2 Standard		B3 Basic	
1	Core	2	Core	4	Core
1.75	GB RAM	3.5	GB RAM	7	GB RAM
	BizTalk Services		50 GB Storage		10 GB Storage
	250 GB Storage		Custom domains / SSL SNI Incl & IP SSL Support		Custom domains
	Up to 20 instances * Subject to availability		Up to 10 instances Auto scale		SSL Support SNI SSL Included
	20 slots Web app staging		Daily Backup		Up to 3 instances Manual scale
	50 times daily Backup		5 slots Web app staging		
	Traffic Manager Geo availability		Traffic Manager Geo availability		

Рис. 7.5. Примеры функциональности на разных уровнях обслуживания

В базовом тарифе доступен только один способ масштабирования приложения — ручное перемещение ползунка с количеством экземпляров (рис. 7.6). Это означает, что если ваше приложение потребует большей вычислительной мощности, вам придется зайти на портал для внесения необходимых изменений.

AVERAGE INSTANCES

1

PREDICTED INSTANCES

2

* Scale by

an instance count that I enter manually

Description

Manual setup means that the number of instances you choose won't change, even if there are changes in load.

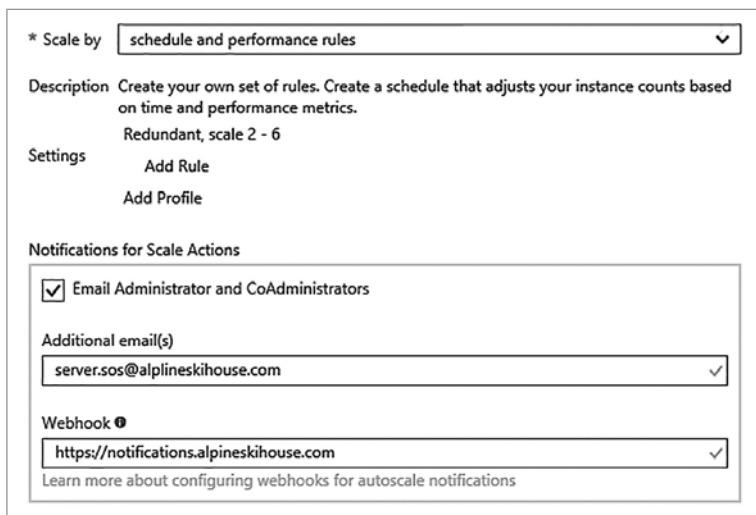
Instances

2

Рис. 7.6. Ручное увеличение количества экземпляров вашего приложения

На стандартном и премиальном уровне обслуживания существует дополнительная возможность горизонтального масштабирования по мере необходимости, с использованием показателей производительности приложения и использования

ресурсов или по возникновению определенных ожидаемых событий. Например, вы можете автоматически нарастить мощности перед ожидаемой повышенной нагрузкой от рекламной кампании, в начале рабочего дня или перед наступлением времени пиковой нагрузки на вашем сайте. Как видно из рис. 7.7, предусмотрена возможность отправки сообщений по электронной почте в зависимости от ограничений производительности или по графику. Вы можете отправлять уведомления или же отправить конечной точке веб-перехватчика данные, которые могут далее использоваться специализированными приложениями мониторинга.



* Scale by schedule and performance rules

Description Create your own set of rules. Create a schedule that adjusts your instance counts based on time and performance metrics.

Redundant, scale 2 - 6

Settings Add Rule Add Profile

Notifications for Scale Actions

☒ Email Administrator and CoAdministrators

Additional email(s) server.sos@alpineskihouse.com

Webhook https://notifications.alpineskihouse.com

[Learn more about configuring webhooks for autoscale notifications](#)

Рис. 7.7. Варианты масштабирования

Очевидно, стандартный уровень обслуживания обходится дороже базового, но эти затраты даже не сравнить с затратами на приобретение и сопровождение сервера. Минимальный стандартный уровень, подходящий для вашей бизнес-модели, позволяет снизить затраты на поддержание продуктивной среды с сохранением возможности автоматического масштабирования под нагрузкой и оплатой только реально используемого времени вычислений.

Если вы только начинаете эксперименты, попробуйте разрешить своему сайту масштабирование между 1 и 2 серверами; это позволит вам оценить, в какое время возрастает нагрузка. Возможно, вам удастся найти способы сэкономить и на более высоких уровнях обслуживания, например при запуске меньшего количества экземпляров, или на более низких уровнях — если вам не нужна вся функциональность более дорогих тарифов.

Также не забудьте, что при использовании модели PaaS существуют и другие возможности, кроме вертикального и горизонтального масштабирования; сейчас мы расскажем о них.

Другие факторы масштабирования

Преимущества масштабирования — лишь часть общей картины, которая может упростить требования к разработке в целом, но за них также приходится платить. Существуют некоторые факторы, которые следует учесть, — и возможно, продумать заново свое решение для сокращения затрат.

Например, возьмем применение кэширования в приложении. Кэширование — один из простейших способов ускорить работу приложения, а кэш в памяти работает невероятно быстро. Но с ростом пользовательской базы может обнаружиться, что кэш поглощает ресурсы App Service, и для сохранения функциональности сайта потребуются масштабирование и дополнительные экземпляры. Более того, если в App Service настроено несколько экземпляров, каждый из них должен поддерживать собственный кэш. Существует простая альтернатива: перейти на распределенный кэш, который одновременно сокращает затраты памяти в приложении и предоставляет централизованный доступ к кэшированным ресурсам.

Другой пример использования ресурсов — журналирование. На каждом уровне приложению выделяется пространство, пропорциональное уплаченной сумме, но следует помнить, что дисковое пространство следует рассматривать как ресурс неустойчивый и не дающий стопроцентных гарантий. Запись в локальный файл журнала на диске имеет ряд последствий. Во-первых, когда вы разбираетесь в причинах ошибки, вам придется просматривать все журнальные файлы со всех серверов, работавших на момент возникновения ошибки. Во-вторых, что более важно, механизмы масштабирования и установки обновлений операционной системы таковы, что перемещение сайта на обновленный экземпляр или его переустановка может привести к потере данных. Поэтому этим причинам следует рассмотреть альтернативный механизм журналирования, который будет более подробно описан в главе 12.

Еще один пример фактора, который может поставить под угрозу работоспособность сайта, — исходящий трафик приложения. Лимит достаточно высок, и все равно существует порог, пересечение которого парализует работу App Service на некоторый период времени. Обычно это происходит в приложениях с большим количеством ресурсов, в которых большие размеры файлов являются нормой, а пользователи создают значительную нагрузку на канал. И снова при попытке взглянуть на задачу с позиций PaaS вы можете перевести эти ресурсы на более подходящий вариант хранения данных в Azure — например, Media Service для потоковой передачи видео и Azure CDN для статических ресурсов.

Итоги

Платформа Microsoft Azure превратилась в мощный набор инструментов, работающих на инфраструктуре, которую нам не нужно поддерживать. Это позволяет разработчикам и техническому персоналу сосредоточиться на результатах работы (а не на том, как эти результаты будут достигаться). Необходимые ресурсы можно легко продублировать в нескольких средах и пользоваться стратегией эластичного масштабирования по мере надобности или допустимых затрат. С готовыми вариантами создания шаблонов ресурсов и методологии построения приложений из готовых «кирпичиков» можно преодолеть многие затруднения, обычно встречающиеся при разработке программных продуктов, и извлечь пользу из надежных, хорошо протестированных сервисов в облаке.

И хотя платформа Azure является предложением из мира облачных технологий от компании Microsoft, она не делает никаких предположений относительно среды разработки и даже используемого стека технологий. В следующей главе мы рассмотрим некоторые возможности разработки, сборки и развертывания приложений на базе ASP.NET Core MVC.

8

Кроссплатформенность

Каждый разработчик — как снежинка: особенное, неповторимое творение Природы. Даниэль была не такой миниатюрной, но вот холодности в отношении MacOS ей было не занимать. Она считала, что вся операционная система — одно сплошное расстройство. В ней было трудно переименовывать файлы, а на экране часто появлялся крутящийся пляжный мяч. Нет уж, она предпочитала свою среду разработки — Windows. С другой стороны, Адриан был сторонником MacOS, а оба Марка не меньше половины рабочего дня проводили за обсуждениями своих любимых дистрибутивов Linux. Марк-2 предпочитает Ubuntu, а Марк-1 восторгался Manjaro Linux. Когда намечалась очередная дискуссия, Даниэль обычно уходила прогуляться, потому что не хотела больше ничего слышать о сравнении Btrfs с Ext4.

Тем не менее это было невероятно. Хотя все разработчики пользовались разными операционными системами и разными текстовыми редакторами, они работали с одной кодовой базой без скольких-нибудь заметных проблем. Время от времени кто-нибудь с файловой системой, игнорирующей регистр символов, использовал ошибочный регистр или разделитель пути, но проведение процессов сборки и в Windows, и в Linux позволило вскоре искоренить подобные проблемы. Тим был действительно заинтересован в том, чтобы новый сайт работал на любой платформе.

Идея о том, что все разработчики могут использовать те машины, которые им больше нравятся, выглядела весьма привлекательно. Также после завершения проекта Parsley ожидалась другие проекты, которые должны были взаимодействовать с сервисами, работающими в Linux. Было бы полезно стандартизировать один язык или хотя бы одну технологию. Возможно, вся эта затея с .NET Core была именно тем, что требовалось в их ситуации.

Возможно, одним из самых замечательных аспектов .NET Core и ASP.NET Core стала полноценная поддержка многих дистрибутивов Linux и MacOS. Она открывает доступ к .NET гораздо большей аудитории разработчиков, включая тех студентов, которые интересуются программированием и не выпускают из рук MacBook. В современном мире операционная система уже не создает ограниче-

ний в том, что касается .NET. И это очень хорошо как с точки зрения новичков, изучающих программирование, так и с точки зрения групп разработки, потому что пропадает одна из причин, по которым в прошлом некоторые команды даже не рассматривали .NET. Для команд разработки на базе Linux и MacOS платформа .NET становилась абсолютно реальным вариантом.

Многие компании используют продуктивные среды на базе Linux по разным причинам. С некоторыми инструментами в Linux работать удобнее, чем в Windows. Например, в поддержке Docker система Windows отстает от Linux, а такие программы управления конфигурацией, как Chef и Puppet, в Linux-версиях все еще лучше, чем в Windows.

Когда была выпущена первая версия Node.js, она работала только в Linux, поэтому особого выбора не было: приложение приходилось запускать в системе, отличной от Windows. Другие инструменты (например, Nginx) тоже работают в Linux лучше, чем в Windows. Есть много ситуаций, в которых система Linux предпочтительна с технической точки зрения, однако возможность запуска инструментов .NET на одной платформе с остальной архитектурой выглядит заманчиво.

Работа в Ubuntu

Проект AlpineSkiHouse изначально создавался в системе Windows с использованием Visual Studio, но это не означает, что мы навсегда ограничены Windows и Visual Studio. Некоторые участники группы чувствуют себя более уверенно, когда их основной операционной системой является Linux. Вместо того чтобы создавать виртуальную машину Windows специально для проекта AlpineSkiHouse, посмотрим, как установить и настроить .NET Core SDK в любимой настольной версии Linux команды AlpineSkiHouse — Ubuntu.

Установка .NET Core

Методы установки .NET Core SDK зависят от операционной системы и хорошо документированы по адресу <http://dot.net>. В случае Ubuntu .NET Core устанавливается с помощью менеджера пакетов apt-get.

Сначала необходимо добавить поставку apt-get, содержащую пакет dotnet:

```
sudo sh -c 'echo "deb [arch=amd64] https://apt-mo.trafficmanager.net/repos/dotnet/
    trusty main"
> /etc/apt/sources.list.d/dotnetdev.list'
sudo apt-key adv --keyserver apt-mo.trafficmanager.net --recv-keys 417A0893
sudo apt-get update
```

Затем установите dotnet:

```
sudo apt-get install dotnet
```

Теперь поддержка .NET Core SDK установлена, а вы можете приступить к сборке приложений .NET на настольных машинах Linux.

Dotnet CLI

Интерфейс командной строки (CLI, Command Line Interface) dotnet является важной составляющей .NET Core SDK. Он используется для компиляции и запуска приложений .NET Core. Хотя многие разработчики, использующие Visual Studio с проектами .NET Core, не знакомы с интерфейсом dotnet CLI, скорее всего, они каждый день используют его. Visual Studio на самом деле использует dotnet CLI для выполнения таких операций, как восстановление пакетов NuGet, компиляция приложения и даже выполнение модульных тестов. Даже если вы собираетесь использовать Visual Studio как основную среду разработки, знать dotnet CLI полезно, потому что этот интерфейс пригодится при создании автоматизированных сценариев сборки. Разработчики, использующие другие операционные системы и облеченные редакторы кода, должны особенно хорошо знать dotnet CLI.

Интерфейс dotnet CLI содержит все необходимое для создания, компиляции и запуска приложений .NET Core. Наиболее часто используемые команды dotnet перечислены в табл. 8.1.

Таблица 8.1. Основные команды dotnet

Команда	Описание
dotnet new	Создает базовый объект .NET
dotnet restore	Восстанавливает зависимости, указанные в проекте .NET
dotnet build	Компилирует проект .NET
dotnet publish	Готовит проект .NET к развертыванию
dotnet run	Компилирует и немедленно выполняет проект .NET
dotnet test	Выполняет модульные тесты
dotnet pack	Создает проект NuGet

Рассмотрим основные этапы создания и выполнения простого приложения Hello World с использованием dotnet CLI. Сначала следует выполнить команду dotnet new. Команда добавляет в текущий каталог два новых файла: файл с информацией о проекте и файл Program.cs с кодом простого приложения Hello World. На рис. 8.1 представлен сгенерированный файл Program.cs.

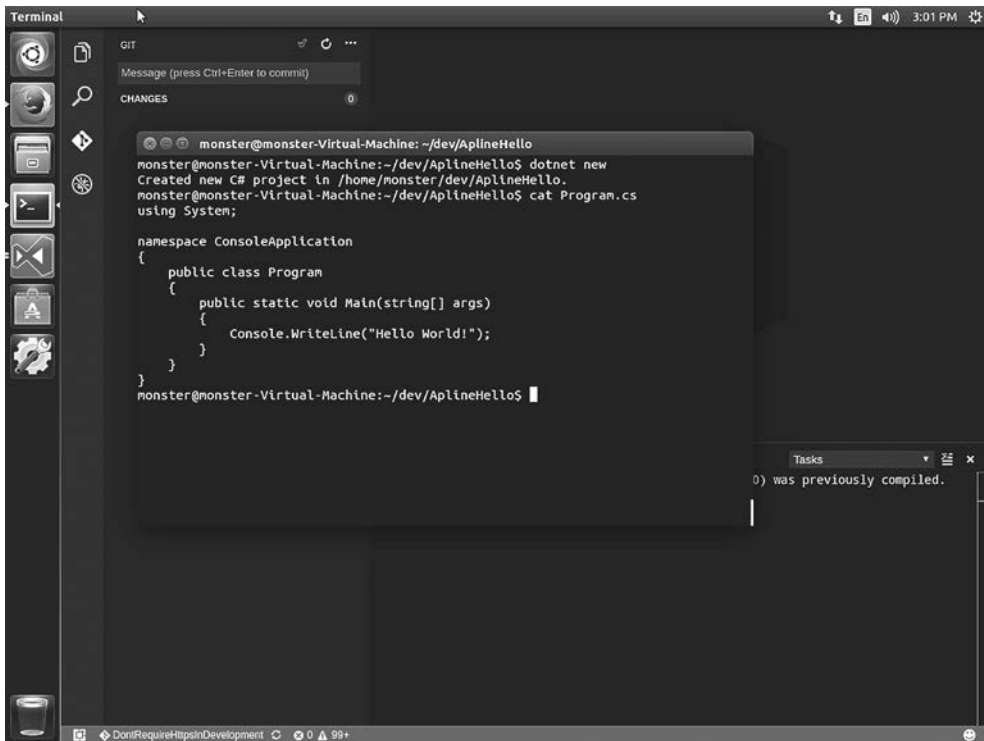


Рис. 8.1. Базовое приложение .NET Core в Ubuntu

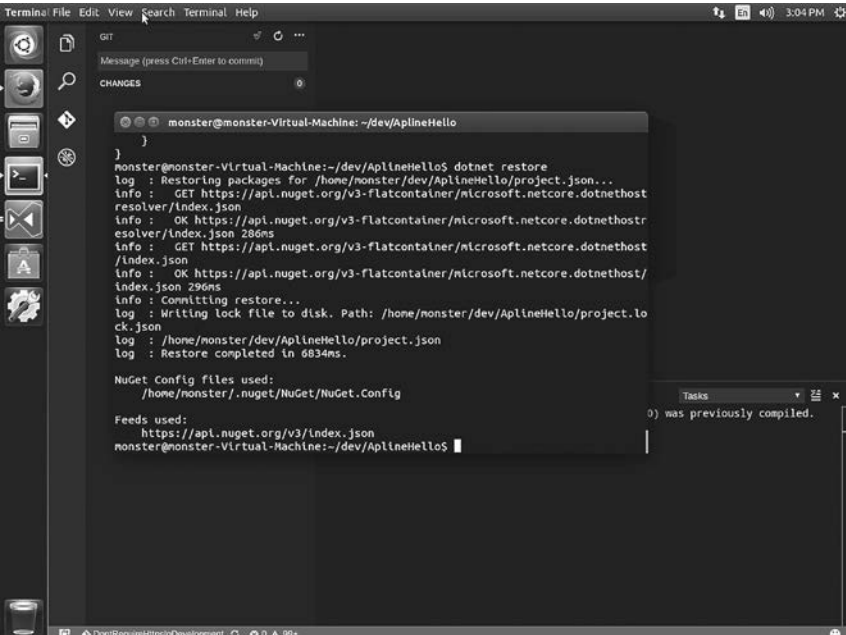
ПРИМЕЧАНИЕ

В командной строке `dotnet` можно создавать проекты разных типов. Чтобы выбрать язык проекта, используйте флаг `-l` и значение `csharp` или `fsharp`. Флаг `-t` определяет тип создаваемого проекта и может принимать значения `console`, `web`, `lib` или `xunittest`. Набор типов проектов куда более ограничен по сравнению с тем, к чему вы привыкли в Visual Studio. При создании нового веб-проекта в Linux присмотритесь к `yeoman` — известному инструменту для построения каркаса веб-приложений, который можно найти по адресу <http://yeoman.io/> или установить через `npm` командой `npm install -g yeoman`.

На следующем шаге выполняется команда `dotnet restore` (рис. 8.2). Команда восстанавливает все пакеты NuGet, необходимые для работы приложения.

Затем приложение можно откомпилировать командой `dotnet build` (рис. 8.3).

Наконец, для запуска приложения используется команда `dotnet run` (рис. 8.4).

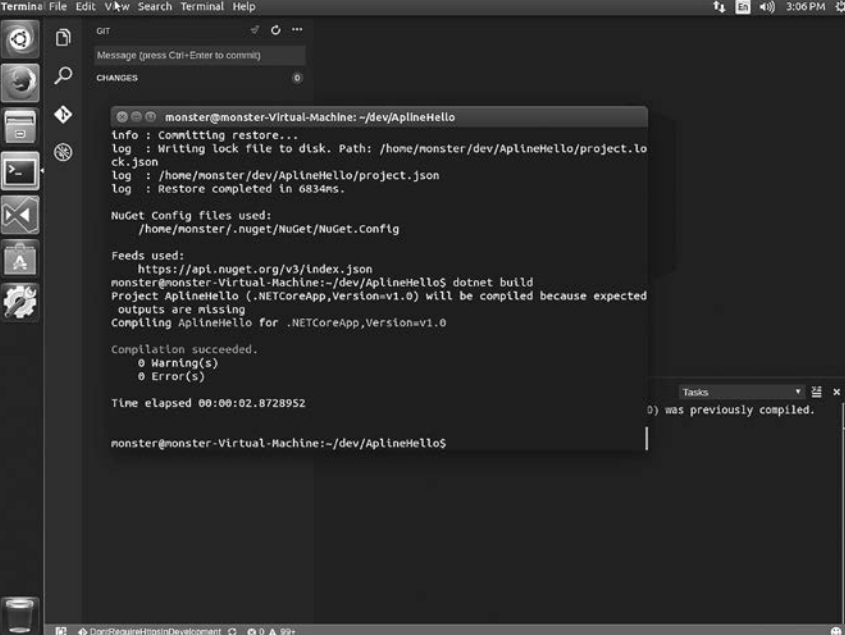


```
monster@monster-Virtual-Machine: ~/dev/AplneHello
}
}
monster@monster-Virtual-Machine:~/dev/AplneHello$ dotnet restore
log : Restoring packages for /home/monster/dev/AplneHello/project.json...
info : GET https://api.nuget.org/v3-flatcontainer/microsoft.netcore.dotnethost
resolver/index.json
info : OK https://api.nuget.org/v3-flatcontainer/microsoft.netcore.dotnethost
resolver/index.json 286ms
info : GET https://api.nuget.org/v3-flatcontainer/microsoft.netcore.dotnethost
/index.json
info : OK https://api.nuget.org/v3-flatcontainer/microsoft.netcore.dotnethost/
index.json 296ms
info : Committing restore...
log : Writing lock file to disk. Path: /home/monster/dev/AplneHello/project.lo
ck.json
log : /home/monster/dev/AplneHello/project.json
log : Restore completed in 6834ms.

NuGet Config files used:
/home/monster/.nuget/NuGet/NuGet.Config

Feeds used:
https://api.nuget.org/v3/index.json
monster@monster-Virtual-Machine:~/dev/AplneHello$
```

Рис. 8.2. Восстановление пакетов nuget в новом проекте



```
monster@monster-Virtual-Machine: ~/dev/AplneHello
info : Committing restore...
log : Writing lock file to disk. Path: /home/monster/dev/AplneHello/project.lo
ck.json
log : /home/monster/dev/AplneHello/project.json
log : Restore completed in 6834ms.

NuGet Config files used:
/home/monster/.nuget/NuGet/NuGet.Config

Feeds used:
https://api.nuget.org/v3/index.json
monster@monster-Virtual-Machine:~/dev/AplneHello$ dotnet build
Project AplneHello (.NETCoreApp,Version=v1.0) will be compiled because expected
outputs are missing
Compiling AplneHello for .NETCoreApp,Version=v1.0

Compilation succeeded.
0 Warning(s)
0 Error(s)

Time elapsed 00:00:02.8728952

monster@monster-Virtual-Machine:~/dev/AplneHello$
```

Рис. 8.3. Выполнение команды dotnet build

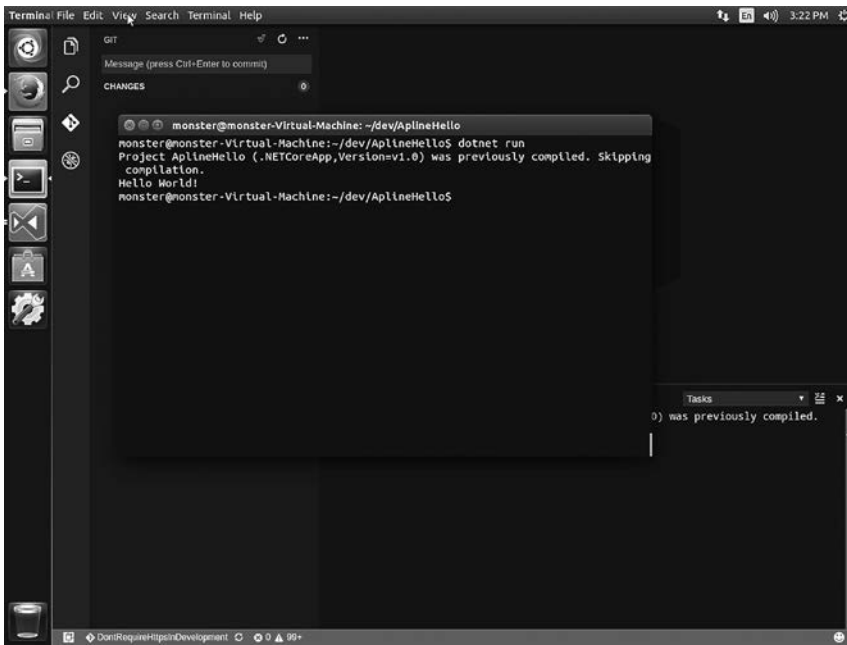


Рис. 8.4. Выполнение нового приложения в Linux

Restore, build, run и test — основные команды, которые используются большинством разработчиков в повседневной работе. Команда `dotnet test` более подробно рассматривается в главе 20 «Тестирование».

Выбор редактора кода

У многих программистов в системах Linux и MacOS есть свой любимый редактор кода. Если вы принадлежите к их числу, то наверняка сможете использовать этот редактор для приложений .NET. Многие разработчики после работы в Visual Studio некомфортно чувствуют себя в прежних минималистичных редакторах из-за отсутствия поддержки IntelliSense. К счастью, эта проблема была решена умными людьми из OmniSharp (<http://www.omnisharp.net/>). OmniSharp создает плагины для многих кроссплатформенных редакторов, что делает возможным использование автозавершения в стиле IntelliSense в проектах .NET. Если у вас есть редактор, который вы хорошо знаете и любите, продолжайте работать в нем, но обязательно проверьте, существует ли плагин OmniSharp, который сделает вашу работу более приятной.

Если вы только ищете редактор, присмотритесь к Visual Studio Code — облегченному кроссплатформенному редактору кода, созданному разработчиками Visual Studio из компании Microsoft. Он предоставляет ту же цветовую подсветку синтаксиса и поддержку IntelliSense, что и Visual Studio, но без всего балласта полно-

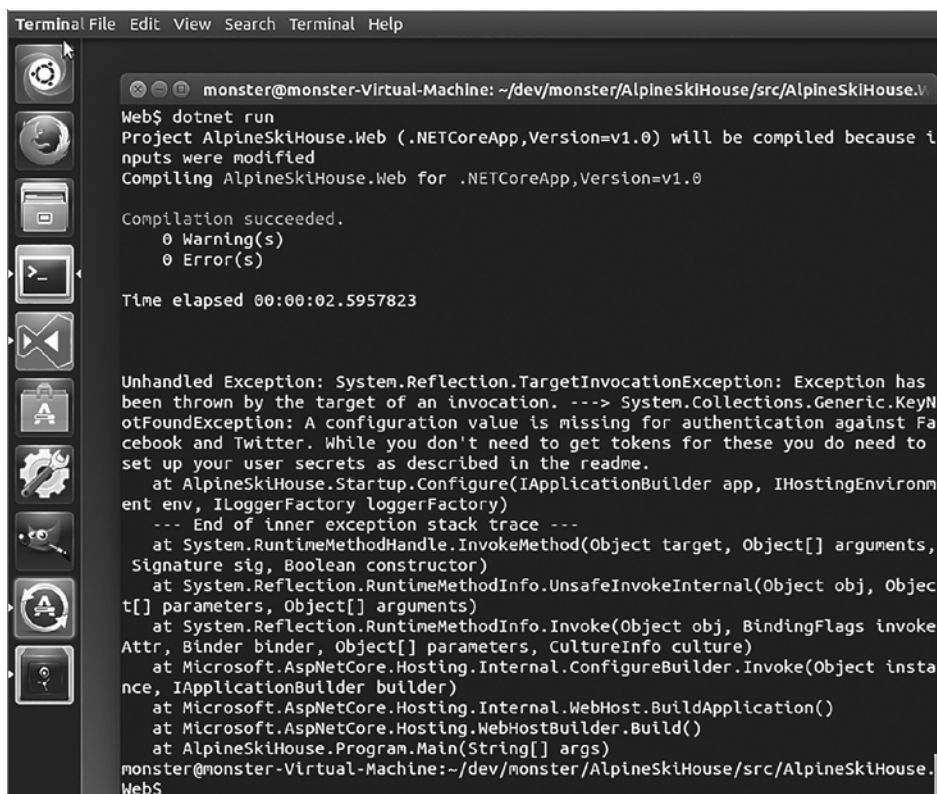
ценной интегрированной среды. Он включает превосходный отладчик, а также может похвастаться богатой экосистемой плагинов. Во всех остальных примерах этой главы будет использоваться Visual Studio Code.

Проект Alpine Ski House в Linux

Итак, установка .NET Core завершена, любимый редактор для разработки .NET настроен — теперь посмотрим, что же необходимо для запуска веб-проекта AlpineSkiHouse на машине разработчика с Ubuntu.

```
$ dotnet build
$ dotnet run
```

Некоторые ошибки на рис. 8.5 показывают, что при запуске не были указаны необходимые скрытые данные пользователей. Следовательно, сначала нужно добавить эти секретные данные командой `dotnet user-secrets`.



```
Terminal File Edit View Search Terminal Help
monster@monster-Virtual-Machine: ~/dev/monster/AlpineSkiHouse/src/AlpineSkiHouse.V
Web$ dotnet run
Project AlpineSkiHouse.Web (.NETCoreApp,Version=v1.0) will be compiled because i
nputs were modified
Compiling AlpineSkiHouse.Web for .NETCoreApp,Version=v1.0

Compilation succeeded.
    0 Warning(s)
    0 Error(s)

Time elapsed 00:00:02.5957823

Unhandled Exception: System.Reflection.TargetInvocationException: Exception has
been thrown by the target of an invocation. ---> System.Collections.Generic.KeyN
otFoundException: A configuration value is missing for authentication against Fa
cebook and Twitter. While you don't need to get tokens for these you do need to
set up your user secrets as described in the readme.
    at AlpineSkiHouse.Startup.Configure(IApplicationBuilder app, IHostingEnvironm
ent env, ILoggerFactory loggerFactory)
    --- End of inner exception stack trace ---
    at System.RuntimeMethodHandle.InvokeMethod(Object target, Object[] arguments,
Signature sig, Boolean constructor)
    at System.Reflection.RuntimeMethodInfo.UnsafeInvokeInternal(Object obj, Objec
t[] parameters, Object[] arguments)
    at System.Reflection.RuntimeMethodInfo.Invoke(Object obj, BindingFlags invoke
Attr, Binder binder, Object[] parameters, CultureInfo culture)
    at Microsoft.AspNetCore.Hosting.Internal.ConfigureBuilder.Invoke(Object insta
nce, IApplicationBuilder builder)
    at Microsoft.AspNetCore.Hosting.Internal.WebHost.BuildApplication()
    at Microsoft.AspNetCore.Hosting.WebHostBuilder.Build()
    at AlpineSkiHouse.Program.Main(String[] args)
monster@monster-Virtual-Machine:~/dev/monster/AlpineSkiHouse/src/AlpineSkiHouse.
Web$
```

Рис. 8.5. Запуск проекта Alpine Ski House в Linux

```
$ dotnet user-secrets set Authentication:Twitter:ConsumerKey "some consumer key"
$ dotnet user-secrets set Authentication:Twitter:ConsumerSecret "some consumer
  Secret"
$ dotnet user-secrets set Authentication:Facebook:AppId "some app id"
$ dotnet user-secrets set Authentication:Facebook:AppSecret "some app secret"
```

Затем необходимо задать переменной `ASPNETCORE_ENVIRONMENT` значение `Development`, потому что приложение загружает скрытые данные пользователей только в режиме `Development`.

```
$ ASPNETCORE_ENVIRONMENT=Development
$ export ASPNETCORE_ENVIRONMENT
```

Теперь можно запустить приложение, верно?

```
$ dotnet run
```

Теперь Kestrel запускается, как и предполагалось, ведет прослушивание по адресу `localhost:5000`, а при загрузке этого URL-адреса в Firefox загружается главная страница. К сожалению, все разваливается при попытке выполнить любое действие, связанное с взаимодействием с базой данных SQL Server (рис. 8.6).

Как упоминается в главе 10 «Entity Framework Core», приложение настроено на использование компонента LocalDB из SQL Server. Такое решение отлично

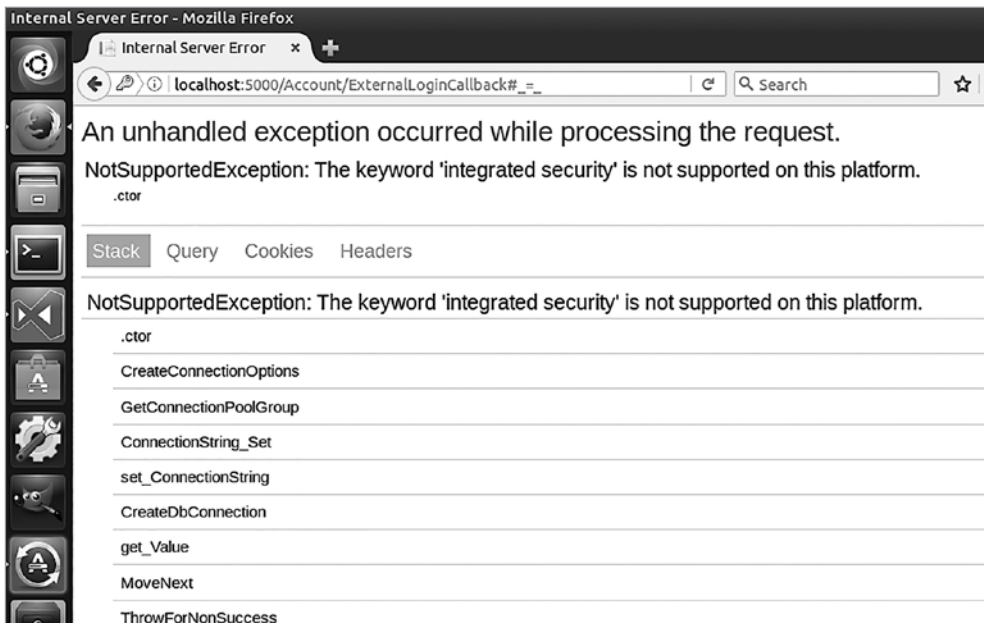


Рис. 8.6. Сообщение об ошибке при подключении к базе данных SQL

работает в Windows, но на момент написания книги поддержка SQL Server в Linux недоступна. И хотя разработчики из Microsoft усердно работают над перенесением SQL Server в Linux, нам понадобится другое решение, которое бы позволило эффективно работать разработчикам Linux и/или MacOS.

В настоящее время лучшее решение основано на предоставлении доступа к экземпляру SQL Server в локальной офисной сети или в облаке. В этом экземпляре SQL Server каждый разработчик получает права на создание и удаление своих баз данных для целей разработки. Не пытайтесь организовать совместный доступ к базам данных для разных разработчиков, потому что это наверняка приведет к проблемам управления версиями. В ходе разработки каждый разработчик должен подключаться к своей уникальной базе данных. Также проследите за тем, чтобы для разработки использовался другой экземпляр SQL Server — не тот, который будет обслуживать рабочие базы данных.

Теперь разработчикам на платформах Linux и macOS достаточно предоставить конфигурацию для переопределения строки подключения LocalDB. Чтобы переопределить конфигурацию, используйте скрытые данные пользователей или добавьте новый файл конфигурации с именем `appSettings.Development.json`.

```
$dotnet user-secrets set ConnectionStrings:DefaultConnection "Data
    Source=tcp:192.168.0.30,3071;
Database=AlpineSkiHouse_Dave;User ID=Dave;Password=Password"
```

appSettings.Development.json

```
{
  "ConnectionStrings": {
    "DefaultConnection": " Data Source=tcp:192.168.0.30,3071;
                          Database=AlpineSkiHouse_Dave;User
ID=Dave;Password=Password"
  }
}
```

А когда все это будет сделано, приложение Alpine Ski House заработает в Ubuntu! Конечно, также необходимо убедиться в правильности прохождения модульных тестов (рис. 8.7).

```
$ dotnet build
$ dotnet test
```

Вообще говоря, наладить работу Alpine Ski House в Linux не так уж сложно. Впрочем, это не простое совпадение — многие проекты попросту не могут работать в Linux или в MacOS. Alpine Ski House использует .NET Core вместо полного фреймворка .NET. .NET Core заслуживает собственного раздела.

```

Terminal
monster@monster-Virtual-Machine: ~/dev/monster/AlpineSkiHouse/test/AlpineSkiHouse
monster@monster-Virtual-Machine:~/dev/monster/AlpineSkiHouse/test/AlpineSkiHouse
.Web.Test$ dotnet restore
log : Restoring packages for /home/monster/dev/monster/AlpineSkiHouse/test/AlpineSkiHouse.Web.Test/project.json...
log : Lock file has not changed. Skipping lock file write. Path: /home/monster/dev/monster/AlpineSkiHouse/test/AlpineSkiHouse.Web.Test/project.lock.json
log : /home/monster/dev/monster/AlpineSkiHouse/test/AlpineSkiHouse.Web.Test/project.json
log : Restore completed in 7360ms.
monster@monster-Virtual-Machine:~/dev/monster/AlpineSkiHouse/test/AlpineSkiHouse
.Web.Test$ dotnet test
Project AlpineSkiHouse.Web (.NETCoreApp,Version=v1.0) was previously compiled. Skipping compilation.
Project AlpineSkiHouse.Web.Test (.NETCoreApp,Version=v1.0) was previously compiled. Skipping compilation.
xUnit.net .NET CLI test runner (64-bit .NET Core ubuntu.14.04-x64)
Discovering: AlpineSkiHouse.Web.Test
Discovered: AlpineSkiHouse.Web.Test
Starting: AlpineSkiHouse.Web.Test
Finished: AlpineSkiHouse.Web.Test
=== TEST EXECUTION SUMMARY ===
AlpineSkiHouse.Web.Test Total: 20, Errors: 0, Failed: 0, Skipped: 0, Time: 0.715s
SUMMARY: Total: 1 targets, Passed: 1, Failed: 0.
monster@monster-Virtual-Machine:~/dev/monster/AlpineSkiHouse/test/AlpineSkiHouse
.Web.Test$

```

Рис. 8.7. Запуск тестов Alpine Ski House в Linux

.NET Core

Иногда мы говорим, что приложение работает на платформе .NET, и считаем, что этим все сказано; однако существуют много разных версий, и даже разных «диалектов» .NET. Полезно для начала подумать, что же составляет ту платформу, которую мы обычно называем .NET. Во-первых, это языки — такие, как C#, F# и VB.NET. Каждый язык имеет собственный стандарт, но все они преобразуются компилятором в общий промежуточный язык IL (Intermediate Language). Компилятор был недавно переписан на C#, сейчас он состоит только из управляемого кода и называется Roslyn. Этот язык выглядит так:

```

IL_0000: ldarg.0          // this
IL_0001: ldfld            int32 class System.Linq.Enumerable/'<CastIterator>d__94'1'
                          <!0/*TResult
lt*/>::'<>1__state'
IL_0006: ldc.i4.s          -2 // 0xfe
IL_0008: bne.un.s        IL_0022
IL_000a: ldarg.0          // this
IL_000b: ldfld            int32 class System.Linq.Enumerable/'<CastIterator>d__94'1'
                          <!0/*TResult*/>:

```



```

:<>l__initialThreadId'
IL_0010: call        int32 [mscorlib]System.Environment::get_
           CurrentManagedThreadId()
IL_0015: bne.un.s    IL_0022
IL_0017: ldarg.0      // this
IL_0018: ldc.i4.0
IL_0019: stfld        int32 class System.Linq.Enumerable/'<CastIterator>d__94'1'
           <!0/*TResu
lt*/>::'<>l__state'
IL_001e: ldarg.0      // this
IL_001f: stloc.0       // V_0
IL_0020: br.s         IL_0029
IL_0022: ldc.i4.0

```

Этот код представляет собой язык ассемблера, который генерируется JIT-компилятором. В соответствии с тенденцией по переработке программ в 64-разрядную форму JIT-компилятор тоже был переписан в виде проекта RyuJIT. Основной целью разработки RyuJIT было повышение эффективности компиляции и генерирование более эффективного 64-разрядного кода.

Все языки .NET могут пользоваться библиотекой базовых классов (BCL). Эта библиотека состоит из тысяч классов для выполнения самых разнообразных операций, от взаимодействия с сервером SMTP до архивации файла в формат ZIP. Пожалуй, широта функциональности библиотеки базовых классов — самое большое преимущество программирования для .NET. Вам не нужны тысячи мелких библиотек (которые нередко встречаются в проектах Node.js); вместо этого разработчики могут положиться на хорошо протестированные, оптимизированные функции в BCL.

Содержимое BCL разрасталось от версии к версии .NET. Например, Silverlight был ограниченной версией фреймворка. Многие классы полного фреймворка в Silverlight отсутствовали или имели другую область применения. .NET Core находится в похожем положении — некоторые классы были исключены или заменены, в результате чего изменилась область применения API. К сожалению, все усилия, направленные на создание новых фреймворков, были разрозненными и хаотичными. Было почти невозможно сказать, какие классы будут доступны на той или иной платформе, и написать код, способный работать на любой платформе.

Предпринимались усилия по созданию портируемых библиотек классов, позволявших разработчикам писать портируемый код, но эти библиотеки быстро стали запредельно сложными. Компания Microsoft запустила собственный проект по нормализации интерфейсов, доступных на многих платформах, включая недавно приобретенные продукты Xamarin.

Этот проект стандартизации получил название .NET Standard. Он определяет ряд уровней совместимости, поддерживаемых разными платформами. Исходная версия определяла уровни с 1.0 по 1.6 и описывала платформы, поддерживающие каждый

уровень. Например, уровень 1.0 предоставляет минимальную область применения API и поддерживается всеми платформами, включая Silverlight. Уровень 1.1 поддерживается всеми платформами, кроме Silverlight. Ниже приведена диаграмма от Microsoft, описывающая доступность API на различных платформах.

.NET Platform	.NET Standard							
	1	1.1	1.2	1.3	1.4	1.5	1.6	2
.NET Core	→	→	→	→	→	→	1	vNext
.NET Framework	→	4.5	4.5.1	4.6	4.6.1	4.6.2	vNext	4.6.1
Xamarin.iOS	→	→	→	→	→	→	→	vNext
Xamarin.Android	→	→	→	→	→	→	→	vNext
Universal Windows Platform	→	→	→	→	10	→	→	vNext
Windows	→	8	8.1					
Windows Phone	→	→	8.1					
Windows Phone Silverlight	8							

Стрелка означает, что платформа поддерживает стандартные возможности, а дополнительные возможности указаны справа. Также обратите внимание на то, что в таблице указана следующая версия каждой платформы. Если, допустим, взглянуть на платформу .NET Framework 4.6.1, поддерживающую .NET Standard 1.4, следует понимать, что .NET Framework 4.6.1 в действительности поддерживает гораздо большую область применения API, чем описано в .NET Standard 1.4, но она гарантированно поддерживает все возможности 1.4 (рис. 8.8).

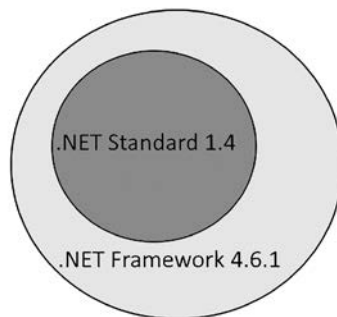


Рис. 8.8. .NET Standard — подмножество API, предоставляемое .NET Framework 4.6.1

Каждая более высокая версия стандарта поддерживает большую область применения, чем предыдущая версия, за исключением 2.0. Как обычно в подобных масштабных проектах стандартизации, возникают аномалии; одной из таких аномалий является версия .NET Standard 2.0, поддерживающая меньшую область применения, чем .NET Standard 1.4 или 1.5 (рис. 8.9).

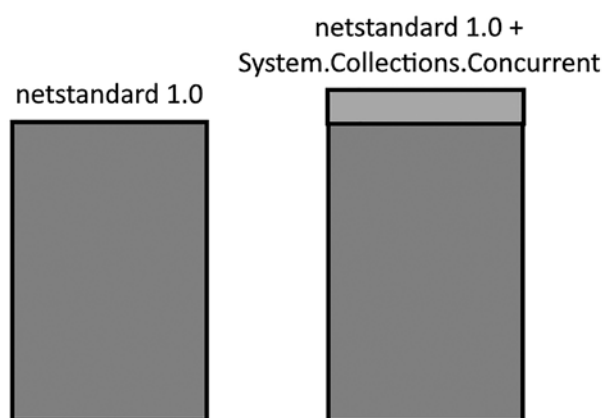


Рис. 8.9. .NET Standard 1.0 и 1.0 с дополнительными пакетами

Предположим, ваше приложение работает в .NET Standard 1.0, но также нуждается в функциональности из `System.Collections.Concurrent`. Есть два возможных решения: во-первых, приложение можно перевести на поддержку .NET Standard 1.1, что добавит поддержку пространства имен параллельных коллекций. Это неуклюжее решение, которое повышает необходимую версию .NET Standard и сокращает количество устройств, на которых может работать ваше приложение. Во-вторых, можно ориентировать приложение на стандарт .NET Standard 1.0 и просто включить пакет NuGet для `System.Collections.Concurrent`. В процессе разработки .NET Standard монолитный фреймворк был разбит на множество пакетов. Собственно, при рассмотрении зависимостей базового приложения в Visual Studio вы увидите в дереве зависимостей, что оно получает целый список мелких пакетов (таких, как `System.Threading.Thread`) для формирования окружения среды .NET Standard. Если добавляемый пакет состоит из управляемого кода, в повышении версии поддерживаемого фреймворка нет необходимости.

Итоги

Работа, направленная на обеспечение некоторой стандартизации .NET API, ведется уже давно, она безусловно нужна и полезна. Можно ожидать, что наличие кроссплатформенной поддержки приведет к принятию .NET в секторах рынка,

которые традиционно не приветствовали технологии .NET. Даже недоброжелатели, которые не приветствуют политику закрытого исходного кода, свойственную для Microsoft, вынуждены пересматривать свои возражения, потому что большая часть стека технологий, включая BCL и компилятор, распространяется с открытым кодом. Простота работы в Linux может привлечь большее количество разработчиков, традиционно работавших в Windows, к OSX и Linux. В любом случае операционная система, под управлением которой работает ваше приложение, уже не является определяющим фактором при выборе .NET.

В следующем разделе мы рассмотрим контейнеры, уделяя основное внимание Docker.

9

Контейнеры

«Сборка работает, все тесты проходят, но я никак не могу заставить эту штуку правильно работать на моей машине, Даниэль». Адриан был близок к отчаянию, Даниэль это видела. Приложение состояло из множества взаимодействующих компонентов. И хотя Даниэль уже несколько лет не работала в таких больших командах, она помнила, как трудно было обеспечить идентичную настройку рабочих сред на разных компьютерах. С усложнением приложений усложняются и их развернутые копии.

«У меня тоже ушла уйма времени, Адриан. Ты заглядывал в документ настройки окружения?»

«В этот кошмар? Он занимает 15 страниц и вечно отстает от жизни. В нем по-прежнему упоминается настройка базы данных, которую мы уже пару недель не используем. Ни у кого нет времени на обновление этого документа».

«Ты прав! — воскликнула Даниэль. — Нужно рассказать об этом Балашу и уговорить его повысить приоритет задачи обновления документации».

Даниэль и Адриан наведались в офис Балаша. «Послушай, мы не можем нормально настроить среду на локальных машинах разработчиков из-за неактуальной документации. Нам нужно выделить больше ресурсов на работу с документацией».

«У нас нет времени все документировать и поддерживать в идеальном состоянии. Нужно назначить одного человека, который отвечает за подготовку среды», — предложил Балаш.

«Это будет самая кошмарная обязанность в конторе, и мне не нравится, как ты на меня смотришь».

«Ты прав, Адриан. Нужно поискать другой способ создания повторяемой, портируемой рабочей среды». — Балаш потер виски.

Слова о повторяемой рабочей среде вызвали в памяти Даниэль какие-то смутные воспоминания.

«Я слышала, что Docker делали как раз для создания рабочей среды и упрощения развертывания».

«А вот это отличная мысль! Лучше у нас будет программа для создания рабочей среды, чем длинный и невразумительный документ. За дело!»

Повторяемые среды

Развертывание приложений и настройка среды для сборки традиционно считались совершенно разными операциями. Рабочая среда создается однократно (например, по образцу среды, использованной для тестирования перед развертыванием), а потом не изменяется. Как выясняется, разработка приложений и настройка среды для сборки — две стороны одной медали. В идеале среда разработки должна как можно точнее воспроизводить рабочую среду, а рабочие и тестовые среды должны создаваться легко и быстро — но тогда это было невозможно. К счастью, понимание того, что задача создания сред должна решаться просто, стимулировало разработку инструментария для таких операций.

Существует много способов повторяемого формирования сред. Самые ранние проекты в этой области были основаны на простом зеркальном копировании жестких дисков программами Symantec Ghost или PowerQuest Drive Image и передаче полученного образа. Загрузка образа целой машины — серьезная задача, но она также заставляет разработчиков использовать те же инструменты, что и другие разработчики, потому что эти инструменты заранее включаются в образ. В прошлом, если разработчики должны были обеспечить поддержку старой версии программного продукта, им приходилось хранить соответствующий образ. Часто бывало проще держать поблизости старые компьютеры для выпуска исправлений. Древние компьютеры прятались в укромных уголках офисов, а к ним прикреплялись записки: «Сборка PSM 1.7 — НЕ ФОРМАТИРОВАТЬ!!!». Записки, конечно, терялись, диски все равно форматировались, что приводило к паническим попыткам воссоздания сред, необходимых для сборки программы десятилетней давности.

Виртуальные машины помогли справиться с некоторыми недостатками применения физических машин для изоляции. Образы создавались намного проще, их версиями было удобнее управлять, а разработчики могли просто запускать собственные инструменты и передавать файлы между локальной и виртуальной машиной. На одной физической машине могли работать несколько виртуальных машин, что сокращало риск изменения специализации критически важной машины. Обеспечиваемый уровень изоляции все еще остается хорошим по сравнению с простым запуском нескольких версий в одной установке операционной системы, потому что вероятность того, что установка обновления на одной виртуальной машине нарушит работоспособность другой виртуальной машины, минимальна.

Изоляция приложений и данных, с тем чтобы они не могли помешать друг другу, всегда была одной из целей операционных систем с момента их создания. Даже без виртуальных машин одно приложение не должно вмешиваться в работу других процессов на машине. Также часто возникает необходимость в изоляции в рамках одного процесса. Программные потоки (threads) обеспечивают некоторую разновидность изоляции, как и область видимости переменных.

Хотя изоляция желательна, за нее приходится платить. С изоляцией всегда связываются некоторые дополнительные затраты, и, как нетрудно догадаться, чем выше уровень изоляции — тем выше затраты. Виртуальные машины обеспечивают высокую степень изоляции, но для этого необходима виртуализация всей физической машины. Драйверы видеоустройств, драйверы дисков, процессоры — все это должно быть виртуализировано. Программные потоки, лежащие на другом конце спектра, требуют минимальных затрат, но их возможности изоляции также невелики. На рис. 9.1 приведено сравнение разных способов изоляции.

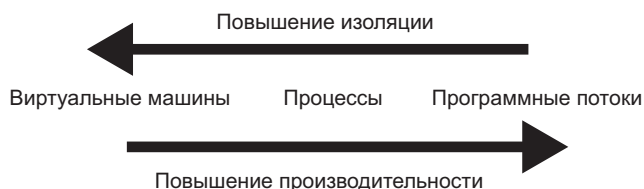


Рис. 9.1. Обратная зависимость между изоляцией и производительностью

Теперь в эту подборку можно добавить контейнеры. И хотя концепция контейнеризации не нова, она по-настоящему обрела самостоятельное существование только за последние пару лет. Контейнеры обеспечивают удачный компромисс по производительности/изоляции между виртуальными машинами и процессами. Контейнеры не виртуализируют всю машину, но предоставляют абстракции над операционной системой, с которыми каждый контейнер выглядит полностью изолированным. Например, если вы работаете в границах одного контейнера, при выводе списка процессов отображается только ограниченный набор процессов, выполняемых внутри контейнера, без процессов пользовательского пространства основной операционной системы или другого контейнера на той же машине. Аналогичным образом файл, записанный на диск внутри контейнера, виден только контейнеру, но остается невидимым вне его. Впрочем, ядро совместно используется всеми контейнерами, поэтому все они должны иметь одинаковую архитектуру и то же ядро, что и управляющая машина. Следовательно, контейнер Linux невозможно запустить на машине с системой Windows или контейнер Windows — на машине Linux без создания промежуточной виртуальной машины.

Хотя такой уровень изоляции выглядит привлекательно сам по себе, более важные характеристики контейнеров относятся к упаковке и компоновке. Файловые системы контейнеров реализуются в виде серии уровней, каждый из которых размещается поверх предыдущего уровня, словно слои в торте «Наполеон».

Контейнер начинается с базового образа; этот образ минимален, он содержит только компоненты, абсолютно необходимые для работы операционной системы. Дистрибутив CoreOS Linux — пример такой среды на базе Linux, а Windows

Nano — пример на базе Windows. Поверх базового образа добавляются различные дополнения. Представьте, что вы строите контейнер для веб-сервера и базы данных. Обычно в контейнере выполняется только один процесс, но пока мы не будем обращать внимания на это обстоятельство.

Базовый уровень состоит из операционной системы, поверх него устанавливается сервер базы данных, веб-сервер и, наконец, данные вашего приложения (рис. 9.2).



Рис. 9.2. Уровни в системе

Из уровней легко строятся полные образы. Если возникнет необходимость в обновлении веб-сервера, берутся два нижних уровня, поверх них размещается новый веб-сервер, а файлы приложений размещаются над ними. Такие минималистичные образы создаются быстро, потому что для них не требуется время загрузки, связанное с полноценными виртуальными машинами. Более того, эти образы достаточно компактны для простого распространения и организации совместного доступа. Если вы работаете в фирме-разработчике, программные продукты которой установлены на многих клиентских серверах, только представьте, насколько вам будет проще жить, если у всех ваших клиентов будет повторяемая, изолируемая среда для работы вашего продукта? Вместо программ установки, сложных в сопровождении, вы можете распространять полные образы для запуска на клиентских машинах.

Ранее мы упоминали о том, что в контейнере должен выполняться только один процесс. Хотя на первый взгляд кажется, что это требование только добавляет лишних затрат (так как эти затраты должны быть связаны с каждым процессом), в действительности оно создает более стабильную, масштабируемую среду. Облегченная природа контейнеров также означает, что на одном оборудовании можно развернуть их намного больше, чем если бы то же оборудование использовалось для запуска виртуальных машин. Изоляция приложений внутри контейнера предоставляет гарантии совместимости общих библиотек и инструментариев. Традиционно система Windows страдала от «кошмара DLL»: разные приложения устанавливали разные глобальные версии библиотек, которые конфликтовали друг с другом. Установка программного обеспечения внутри контейнера снимает эту проблему.

Контейнеры не обязаны ограничиваться одной машиной. Развертывание большого приложения может требовать десятков разных сервисов, которые развертываются в разном масштабе. Например, могут потребоваться десятки веб-серверов, полдюжины серверов поиска и два сервера баз данных. Создание таких сред на физическом оборудовании всегда создавало трудности, тогда как с контейнерами ресурсное распределение серверов становится скорее программной, нежели аппаратной проблемой. Вместо того чтобы выделять машину под веб-сервер, ее можно просто сделать управляющей машиной контейнера. Система управления контейнерами располагает информацией о пуле доступных для нее серверов. Читая функции конфигурации, она может развернуть контейнеры на серверах в произвольной конфигурации. На следующей диаграмме показаны контейнеры, развернутые в группе физических машин (рис. 9.3).

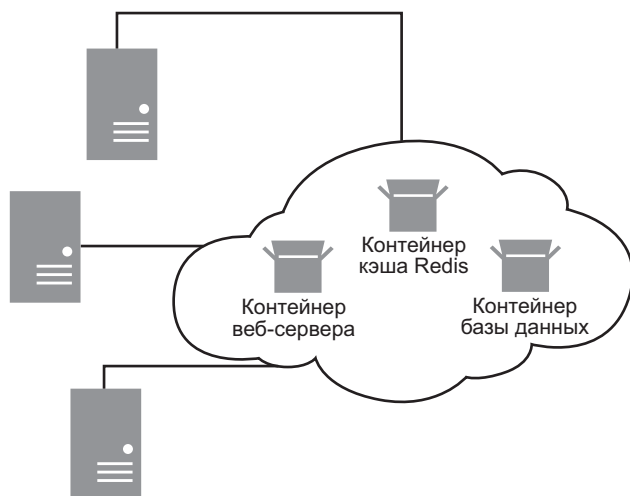


Рис. 9.3. Контейнеры, развернутые в системе управления контейнерами на нескольких физических серверах

Так как контейнеры могут быстро создаваться и уничтожаться, система управления может принимать решения о перемещении контейнеров между физическими машинами при сохранении работоспособности приложения. Контейнеры не знают, где они выполняются, а любые коммуникации между контейнерами находятся под контролем системы управления.

В контейнерных средах могут строиться сети, которые описываются определениями на программном уровне. Например, контейнер веб-сервера может иметь возможность обмениваться данными с контейнером базы данных и контейнером кэша Redis, но контейнеру базы данных не нужно обмениваться данными с контейнером Redis. Кроме того, для внешних подключений могут использоваться только порты веб-сервера и, возможно, некоторые порты управления (рис. 9.4).

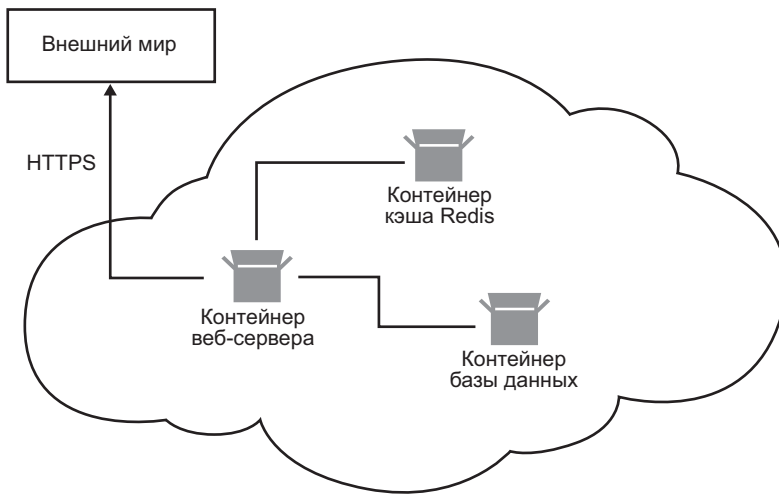


Рис. 9.4. Закрытая сеть контейнеров, внешний доступ к которой возможен только по протоколу HTTPS

Возможность формирования на программном уровне сред, которые могут легко масштабироваться, перемещаться или воспроизводиться, воистину бесценна. Контейнеры и управляющая система предоставляют преимущества воспроизводимой облачной инфраструктуры, но с сохранением портируемости между облачной и локальной инфраструктурой. На локальной машине разработчика может работать полноценная среда, что повышает надежность разработки и тестирования с минимальными проблемами.

Docker

Возможно, вы заметили, что до настоящего момента в этой главе не упоминалась никакая конкретная контейнерная технология. Как и любые хорошие технические идеи, она существует в нескольких реализациях.

Docker — самая известная из всех контейнерных технологий. Docker предоставляет целостный инструментарий на основе средств виртуализации, уже существующих внутри ядра Linux. Также Docker предоставляет общий формат для передачи образов. Docker Registry — специальное место для сохранения и загрузки образов, что упрощает их создание. Docker также может использоваться для запуска контейнеров Windows в Windows Server 2016 и в Anniversary Update для Windows 10.

Наряду с Docker также существует похожая технология rkt (произносится «рокет»), разработанная группой CoreOS. Rkt предоставляет стандартный открытый формат контейнеров, не требующий запуска специального демона. Модель безо-

пасности rkt предотвращает похищение образов и имеет менее монолитную архитектуру, чем Docker, потому что rkt позволяет заменять компоненты по мере изменения технологий. Как rkt, так и Docker строится на общей функциональности cgroups, встроенной в ядро Linux. В принципе реализация rkt лучше, но, как и в случае с Betaux, более совершенная технология не всегда побеждает.

В этом разделе вы узнаете, как запустить приложение ASP.NET Core в контейнере Linux. Для начала нужно установить Docker. Если вы используете Windows, вероятно, лучше установить версию Docker для Windows, которая находится по адресу <http://www.docker.com/products/docker#/windows>. Docker использует гипервизор HyperV для запуска виртуальной машины Linux в Windows. Именно на этой машине запускается демон Docker. Контейнерами здесь являются «родные» контейнеры Linux, которые могут быть установлены на сервере Linux в облаке или в вашем центре обработки данных. Конечно, вы также можете работать с контейнерами Windows (эта тема рассматривается в следующем разделе). Концепции при этом не изменяются, поэтому если вы намерены запускать контейнеры Windows — дочитайте этот раздел до конца, прежде чем двигаться дальше.

После установки Docker можно создать первый образ. Команда Docker, установленная в Windows, соединяется с демоном Docker на виртуальной машине. Эта программа может подключиться к любому демону Docker как на локальной виртуальной машине, так и на находящейся в облаке. Docker для Windows автоматически настраивает переменные окружения для подключения к демону на виртуальной машине.

Компания Microsoft любезно построила образ Docker с предустановленной последней версией .NET Core. Используйте его в качестве базового образа для Dockerfile. На самом деле у Microsoft есть целый набор образов Docker с разными комбинациями SDK и базовых операционных систем. Например, образ *microsoft/dotnet:1.0.0-preview2-sdk* содержит новейшую (на момент написания книги) версию SDK с операционной системой Linux, а образ *microsoft/dotnet:1.0.0-preview2-nanoserver-sdk* содержит новейшую версию SDK на базе Windows Nano. Полный список контейнеров, предоставляемых Microsoft, доступен по адресу <https://hub.docker.com/r/microsoft/dotnet/>.

Начните с пустого текстового файла в корне проекта и введите следующую информацию:

```
FROM microsoft/dotnet:1.0.0-preview2-sdk

COPY . /app

WORKDIR /app

RUN ["dotnet", "restore"]

RUN ["dotnet", "build"]
```

```
RUN ["dotnet", "ef", "database", "update"]
```

```
EXPOSE 5000/tcp
```

```
ENTRYPOINT ["dotnet", "run", "--server.urls", "http://0.0.0.0:5000"]
```

Файл содержит инструкции, которые используются Docker при построении образа для развертывания. Первая строка определяет базовый образ, на основании которого строится итоговый образ. В данном примере указана конкретная версия, но если вы предпочитаете работать с новейшей версией, можно использовать запись `microsoft/dotnet:latest`. Следующая строка копирует содержимое текущего каталога в папку `/apps` образа. Конечно, в Linux существует стандарт для размещения файлов, но поскольку эти образы являются временными, мы игнорируем правила и размещаем файлы в корне — словно студент, получивший копию Red Hat 5.2 на записанном компакт-диске. Какие воспоминания...

В качестве рабочего каталога назначается только что скопированная папка `/app`. В этой папке проводится восстановление пакетов (команда `dotnet restore`), после чего сборка приложения приводит его в работоспособное состояние. Далее разрешаются подключения по TCP к контейнеру через порт 5000. Наконец, команда `dotnet` запускает копию Kestrel на порту 5000. В директиве `ENTRYPOINT` указывается команда, которая должна выполняться при запуске контейнера, а все остальное выполняется во время сборки.

Вывод команды получается достаточно длинным, потому что в него включены обширные данные восстановления пакетов. В конце сборки выводится следующий фрагмент:

```
Step 8 : ENTRYPOINT dotnet run --server.urls http://0.0.0.0:5000
---> Running in 78e0464bc9fe
---> 3a83db83863d
Removing intermediate container 78e0464bc9fe
Successfully built 3a83db83863d
```

Завершающая алфавитно-цифровая строка представляет собой идентификатор контейнера.

После создания файла `Dockerfile` выполните следующую команду:

```
docker build -t aplineski
```

Команда создает образ Docker с именем *alpineski*. Впрочем, если вы собираетесь распространять образы, имя должно быть более конкретным.

Созданный образ помещается в хранилище на машине. Для управления хранилищем используются соответствующие команды. Чтобы вывести список образов, введите команду:

```
docker images
```

Чтобы удалить образ, используйте команду:

```
docker rmi <идентификатор>
```

После того как контейнер заработает, в него можно внести изменения и закрепить измененный контейнер в образе, который может использоваться как базовый в будущем. Это делается следующей командой:

```
Docker commit <идентификатор> <имя образа>
```

В большинстве случаев для создания образов следует использовать Dockerfile (вместо закрепления модифицированного работающего контейнера), потому что этот способ лучше воспроизводится.

Чтобы запустить созданный вами контейнер, введите команду:

```
docker run -d -t -p 5000:5000 3a83db83863d
```

Команда запускает фоновый контейнер, которому перенаправляются обращения к порту 500 управляющей машины Docker. При запуске контейнера выводится еще один очень длинный хеш-код, который выглядит примерно так:

```
731f082dc00cce58df6ee047de881e399054be2022b51552c0c24559b406d078
```

Команда `docker ps` выводит более короткую версию этого хеш-кода и список всех работающих контейнеров.

```
docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS
PORTS         NAMES
731f082dc00c   3a83db83863d   "dotnet run --server."   19 seconds ago Up 20
seconds
0.0.0.0:5000->5000/tcp   angry_shirley
```

Первое поле (`container id`) содержит сокращенную форму идентификатора. Чтобы просмотреть вывод контейнера, воспользуйтесь командой `Docker logs`:

```
docker logs 731f082dc00c
Project app (.NETCoreApp,Version=v1.0) was previously compiled. Skipping
  compilation.
info: Microsoft.Extensions.DependencyInjection.DataProtectionServices[0]
      User profile is available. Using '/root/.aspnet/DataProtection-Keys' as key
        repository;
keys will not be encrypted at rest.
Hosting environment: Development
Content root path: /app
Now listening on: http://*:5000
Application started. Press Ctrl+C to shut down.
```

Вероятно, этот фрагмент покажется вам знакомым, потому что он завершает вывод команды `dotnet run`. После завершения работы с контейнером его можно завершить командой:

```
docker stop
```

Если вы просто хотите приостановить выполнение контейнера, введите команду:

```
docker pause
```

Работа приостановленного контейнера продолжается командой:

```
docker unpause
```

Хранение данных — другой важный аспект контейнеров Docker. Применение контейнеров выглядит абсолютно логично для временных данных и функциональности. Например, веб-сервер не имеет состояния, поэтому завершение и уничтожение одного контейнера и запуск другого не окажет особого влияния на процесс отображения веб-страниц. С другой стороны, сервер базы данных должен постоянно управлять состоянием информации. Для этого можно смонтировать каталог машины, на которой работает Docker, в работающем образе. Это позволит вам завершить контейнер и создать другой с присоединением того же тома без каких-либо проблем. При запуске контейнера флаг `-v` монтирует указанный том, например:

```
docker run -d -t -v /var/db:/var/postgres/db -p 5000:5000 3a83db83863d
```

Команда монтирует каталог `/var/db` в контейнере как `/var/postgres/db`. Смонтированные каталоги даже могут быть доступными через iSCSI, а значит, том можно смонтировать напрямую из SAN с обеспечением желательной избыточности.

Файл Docker для Alpine Ski будет более сложным, чем созданный вами файл по умолчанию. Есть ряд аспектов, которые необходимо учитывать. Есть несколько разных контекстов данных, каждый из которых должен обрабатываться независимо. Также необходимо принять во внимание переменные Twitter и Facebook, которые могут задаваться с использованием переменных окружения. Наконец, необходимо задать значение переменной `ASPNETCORE_ENVIRONMENT`, чтобы избежать перенаправления на несуществующую SSL-версию сайта.

```
FROM microsoft/dotnet:1.0.0-preview2-sdk
```

```
COPY . /app
```

```
WORKDIR /app
```

```
RUN ["dotnet", "restore"]
```

```
RUN ["dotnet", "build"]
```

```
RUN ["dotnet", "ef", "database", "update", "--context=ApplicationUserContext"]
```

```
RUN ["dotnet", "ef", "database", "update", "--context=PassContext"]
```

```
RUN ["dotnet", "ef", "database", "update", "--context=PassTypeUserContext"]
```

```
RUN ["dotnet", "ef", "database", "update", "--context=ResortContext"]
```

```
RUN ["dotnet", "ef", "database", "update", "--context=SkiCardContext"]
```

```
EXPOSE 5000/tcp
```

```
ENV Authentication:Facebook:AppSecret FacebookAppSecret
ENV Authentication:Facebook:AppId FacebookAppId
ENV Authentication:Twitter:ConsumerSecret TwitterSecret
ENV Authentication:Twitter:ConsumerKey TwitterKey

ENV ASPNETCORE_ENVIRONMENT Development

ENTRYPOINT ["dotnet", "run", "--server.urls", "http://0.0.0.0:5000"]
```

Организовать совместное использование контейнеров на базе Docker сложнее; собственно, это одна из областей, в которых `rkt` проявляет себя во всем блеске. С `rkt` можно экспортировать образы и загрузить их прямо на веб-сервер, откуда их могут брать другие. Docker также позволяет экспортировать контейнеры с использованием команды `Docker save`, но эти файлы приходится импортировать вручную в других экземплярах Docker. Чтобы организовать общий доступ к образам Docker, необходимо запустить хранилище Docker командой `Docker image`. Также возможно отправить образ в открытое хранилище Docker с именем `Hub`. Естественно, это означает, что образ будет доступен для всех желающих. Некоторые фирмы-разработчики распространяют закрытые версии хранилища на коммерческой основе.

Контейнеры Windows

Выдающийся успех контейнеров в Linux привел к включению контейнерной технологии в Windows Server 2016 и Windows 10 Anniversary edition. Однако по умолчанию она не устанавливается, и ее необходимо подключать отдельно. Более того, если Windows у вас работает на виртуальной машине, успех не гарантирован.

Для начала необходимо подключить службу контейнеров и гипервизор HyperV. На этом шаге может произойти сбой, если ваша версия Windows несовместима с контейнерами. К сожалению, на этом шаге потребуются перезапуск системы. Когда-нибудь будет изобретена операционная система, конфигурация которой может меняться без обязательного перезапуска, но, очевидно, это время еще не пришло.

```
Enable-WindowsOptionalFeature -Online -FeatureName containers -All
Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Hyper-V -All
Restart-Computer -Force
```

СОВЕТ

Если вы используете Windows 10, поддержку контейнеров можно установить из диалогового окна «Приложения и возможности» (Programs and Features) панели управления. Но разве не удобнее работать в командной строке?

После завершения перезагрузки необходимо загрузить и настроить Docker. Продукт распространяется в виде простого zip-файла, который следует распаковать и добавить в переменную окружения Path.

```
Invoke-WebRequest "https://master.dockerproject.org/windows/amd64/docker-
1.13.0-dev.zip"
-OutFile "$env:TEMP\docker-1.13.0-dev.zip" -UseBasicParsing
Expand-Archive -Path "$env:TEMP\docker-1.13.0-dev.zip" -DestinationPath
$env:ProgramFiles
# For quick use, does not require shell to be restarted. $env:path += ";
c:\program files\docker"
# For persistent use, will apply even after a reboot. [Environment]::
SetEnvironmentVariable("Pa
th", $env:Path + ";C:\Program Files\Docker", [EnvironmentVariableTarget]::Machine)
```

После установки Docker регистрируется как служба и запускается.

```
dockerd --register-service
Start-Service Docker
```

Если на машине установлено более одного экземпляра Docker, обязательно укажите правильный экземпляр для использования с контейнерами Windows. Процедура установки добавит инструменты Docker в переменную пути, но поскольку новая информация будет располагаться в конце, может оказаться, что она не будет выполнена.

Теперь вы можете протестировать свое решение на базе контейнеров Windows, создав для него новый файл Dockerfile. Большинство существующих файлов Dockerfile может использоваться без изменений, потому что команды для разных операционных систем практически совпадают.

```
FROM microsoft/dotnet:1.0.0-preview2-windowsservercore-sdk
COPY . /app
WORKDIR /app
RUN ["dotnet", "restore"]
RUN ["dotnet", "build"]
RUN ["dotnet", "ef", "database", "update", "--context=ApplicationUserContext"]
RUN ["dotnet", "ef", "database", "update", "--context=PassContext"]
RUN ["dotnet", "ef", "database", "update", "--context=PassTypeUserContext"]
RUN ["dotnet", "ef", "database", "update", "--context=ResortContext"]
RUN ["dotnet", "ef", "database", "update", "--context=SkiCardContext"]
EXPOSE 5000/tcp
ENV Authentication:Facebook:AppSecret FacebookAppSecret
```

```
ENV Authentication:Facebook:AppId FacebookAppId
ENV Authentication:Twitter:ConsumerSecret TwitterSecret
ENV Authentication:Twitter:ConsumerKey TwitterKey
ENV ASPNETCORE_ENVIRONMENT Development
```

```
ENTRYPOINT ["dotnet", "run", "--server.urls", "http://0.0.0.0:5000"]
```

Обратите внимание: этот контейнер создан на базе Windows Server Core. Похоже, в настоящее время сборка контейнеров Windows Nano работает только из самих контейнеров Nano, если не использовать менее эффективную контейнеризацию Hyper-V. Файл Dockerfile должен работать в большинстве экземпляров Windows. Контейнеры Windows делятся на две разновидности: контейнеры Hyper-V и контейнеры Windows Server. Контейнеры Windows Server более компактны и приближены к контейнерам на базе cgroups в Linux, а ядро используется совместно всеми контейнерами в системе. Контейнеры Hyper-V существуют внутри виртуальной машины. Они лучше изолированы, но уступают контейнерам Windows Server по эффективности.

ПРИМЕЧАНИЕ

Если у вас возникнут проблемы с контейнерами Docker в вашей локальной установке Windows, опробуйте их в Azure с контейнером Windows, заранее установленным с Docker. Правда, вы не сможете работать в нем с контейнерами Hyper-V, потому что он сам является контейнером Hyper-V.

Контейнерные технологии Windows еще очень молоды. На момент написания книги система Windows Server 2016 еще не выпущена, а поддержка контейнеров в Windows 10 фактически находится на уровне бета-версии, хотя формально и не имеет такого статуса. По мере развития технологий и повышения важности контейнеров усилия, потраченные на их совершенствование и повышение надежности, только возрастают. И хотя прогнозирование будущего — дело неблагодарное, можно с уверенностью сказать, что лет через пять контейнеры станут стандартным механизмом развертывания не только для веб-приложений, но и для настольных приложений. А пока это время не настало, нам приходится решать проблемы, связанные с запуском контейнеров в рабочей среде.

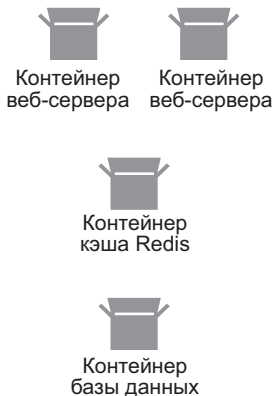
Docker в рабочей среде

На момент написания книги выполнение Docker в рабочей среде с Docker Swarm, официальным инструментом управления кластерами Docker, работает нестабильно. Есть много сообщений о сетевых сбоях и ошибках передачи данных между узлами; впрочем, Docker Swarm еще находится на ранней стадии развития. Существуют альтернативные решения по распределенной поддержке Docker, но они не используют официальные компоненты Swarm.

Apache Mesos — система управления кластерами, целью которой является попытка абстрагироваться от центра обработки данных и интерпретировать ситуацию так, словно весь центр обработки данных находится на одном компьютере. С момента выпуска первой версии в 2009 году программа получила широкое распространение и даже была задействована в технологии распространения контейнеров Azure (<https://azure.microsoft.com/en-us/documentation/videos/azurecon-2015-deep-dive-on-the-azure-container-service-with-mesos/>). Ее поддержкой занимается коммерческая организация Mesosphere. По сути Mesos является системой распределенного планирования, но расширение позволяет легко использовать ее для распределения контейнеров. Вероятно, самым известным инструментом для управления контейнерами в Mesos является Marathon.

Компания Google, которая не привыкла отставать от других, создала и опубликовала аналогичный инструмент с открытым кодом, который называется Kubernetes. И если управление контейнерами было встроено в Mesos спустя много времени после создания системы, программа Kubernetes изначально создавалась для управления контейнерами. Kubernetes делится на фундаментальные структурные элементы: капсулы, метки, контроллеры, сервисы и узлы. Капсула (pod) представляет собой совокупность контейнеров, работающих на одной машине. Если у вас есть контейнеры, которые должны работать поблизости друг от друга, возможно, вам стоит организовать их в капсулу. Метки (labels) представляют собой пары «ключ — значение», которые могут назначаться любому узлу или капсуле. Контроллеры отвечают за согласование фактического состояния системы с желательным. Например, представьте, что у вас имеются два контейнера веб-серверов, контейнер базы данных и контейнер Redis, но в фактическом состоянии задействован только один веб-сервер (рис. 9.5).

Желательное состояние



Фактическое состояние



Рис. 9.5. Желательное и фактическое состояние — Kubernetes выполняет необходимые операции для приведения фактического состояния к желательному

При получении нового желательного состояния система Kubernetes выполняет цикл согласования до тех пор, пока фактическое состояние системы не совпадет с желательным. Таким образом вы избавляетесь от необходимости вручную проводить сравнения желательного состояния с фактическим. Kubernetes вычисляет, какие операции нужны для приведения системы к нужному состоянию.

Наконец, сервис (service) в Kubernetes представляет собой коллекцию капсул, совместно работающих для предоставления определенного блока функциональности. Помните, что проектировщики Kubernetes ориентировались на концепцию микросервисов, так что сервисы могут быть достаточно мелкими. Количество развернутых сервисов является точкой масштабирования, позволяя наращивать количество микросервисов с ростом спроса. Сервисы могут предоставлять конечные точки с возможностью маршрутизации, доступ к которым может предоставляться другим сервисам или за пределами кластера.

В облаке

Существует много вариантов размещения контейнеров в облаке. У всех основных поставщиков облачных сервисов существуют те или иные решения, относящиеся к облачным вычислениям. Google Compute Engine предоставляет сервис облачных вычислений на базе контейнеров, использующий управляющее ядро Kubernetes той же компании Google. У Google больше опыта контейнеризации, чем у любых других поставщиков, а их предложение выглядит наиболее зрелым.

Реализация контейнеров от Google ближе к тому, что мы обычно ожидаем от предложений PaaS в облаке. Нет необходимости создавать собственный кластер виртуальных машин. Вместо этого ваши компьютеры работают как часть общего компьютерного облака. Предложения AWS и Azure в настоящее время требуют, чтобы конечный пользователь создал собственную группу виртуальных машин для построения контейнерного кластера.

В сфере облачных технологий в настоящее время идет яростная конкуренция. Каждый крупный поставщик облачных услуг изо всех сил старается привлечь на свою сторону как можно больше компаний, которые еще не перешли на облачные технологии. Теоретически перемещение между облаками должно проходить просто, но на деле трудно представить, что заставит большинство компаний выполнить такой переход; поэтому так важно привлечь несколько оставшихся компаний и связать их с облачными технологиями конкретного производителя. Хотя контейнеризация ослабляет такую привязку к производителю, несомненно, в будущем станут предприниматься значительные усилия по упрощению использования контейнеров.

Итоги

Может показаться, что технология контейнеризации еще находится в начальной стадии своего развития, но она очень быстро растет (особенно если верить всей шумихе по этому поводу). Компания Microsoft вложила значительные средства в контейнеризацию своих серверов и настольных операционных систем, а также в Azure. В самом деле, контейнеризация означает фундаментальные изменения в подходе к сборке и развертыванию сервисов. Она способствует разделению приложений на меньшие узкоспециализированные фрагменты. Конечно, контейнеры — не панацея, и в некоторых областях прекрасно работает старый, монолитный подход к проектированию приложений. Даже если ваша архитектура не требует разбиения на меньшие части, контейнеры могут пригодиться для формирования последовательной структуры сборки и тестирования ваших приложений. Запуск разработчиками собственных экземпляров контейнеров может ускорить тестирование и обеспечить согласованность между их средой и средами других участников команды.

Безусловно, вашей группе стоит потратить неделю своего времени на то, чтобы понять, как контейнеры помогут справиться с текущими «узкими местами» разработки. А в следующей главе вы узнаете, как организовать простой и эффективный доступ к базе данных с использованием Entity Framework.

10

Entity Framework Core

Работа над проектом Parsley, новой системой продажи абонементов на подъемники для горнолыжного курорта Alpine Ski House, продвигалась вполне успешно. Даниэль чувствовала, что участники команды понемногу притираются друг к другу и их связывает общая цель. Как раз в это время Марк-1 своим привычным голосом запел:

«Дело вовсе не в данных, данных, данных,

Нам не нужны твои данные, данные, данные.

Мы просто хотим поставить мир на лыжи,

И забудьте про индекс».

Даниэль покосилась на него. «Это что, Джесси Джей?» — спросила она.

«Так точно, — ответил Марк-1. — Она — Бах XXI века. Но вообще-то я думал о данных. Нам нужно разместить набор данных в базе, а потом снова извлечь их. Казалось бы, проблема уже давно должна быть решена, но я до сих пор пишу в своем коде команды SQL».

Даниэль знала, что не для каждой задачи доступа к данным требуется идеальное решение, и в некоторых ситуациях низкоуровневый код SQL мог быть лучшим из возможных вариантов. «Мы тут поговорили о хранилищах NoSQL и решили, что никто из нас не обладает достаточным опытом, чтобы использовать их в этом проекте. Остается хранилище данных SQL или, вероятно, SQL Server, раз уж мы размещаемся в Azure, — сам понимаешь, путь наименьшего сопротивления. Я использую Entity Framework для большинства операций с данными».

«Вот как, — задумался Марк-1. — Я никогда не использовал Entity Framework, но звучит так... солидно. Это что-то связанное с Enterprise Java Beans? И мне придется создавать класс `SkiPassEntityFactoryBeanFactoryStrategy`? Не думаю, что у нас на это есть время, Даниэль».

Даниэль засмеялась, этот парень нравился ей все больше. «Не беспокойся, Entity Framework Core на самом деле является результатом многолетних экспериментов с Entity Framework (EF), хотя по его нумерации версий это не заметно. С помощью EF данные можно извлечь эффективно и просто. Хватай кресло, и я тебе кое-что покажу, пока ты снова не запел».

Можно с уверенностью сказать, что данные являются самым ценным аспектом бизнес-приложения. Не удивительно, что довольно большой процент кода приложения обычно описывает хранение и выборку данных. Сюда относится чтение данных из хранилища, отслеживание изменений и сохранение изменений в том же хранилище. За прошедшие годы появилось несколько технологий .NET, упрощающих эту задачу. Не много найдется решений, порождающих столь полярные мнения, как выбор технологии и методологии хранения и выборки данных в приложениях (не считая разве что вечного вопроса «табуляция или пробелы?»).

Entity Framework (EF) Core — новейшая технология работы с данными, созданная группой разработки ASP.NET; именно этот фреймворк рекомендуется использовать при создании приложений ASP.NET Core. EF Core относится к категории ORM (Object Relational Mapper, «объектно-реляционное отображение»); это означает, что вы избавляетесь от сложностей преобразования данных из реляционного хранилища и объектно-ориентированной предметной модели вашего приложения. EF Core — не единственная ORM-технология, доступная для приложений .NET Core. Dapper — популярный фреймворк «микро-ORM», а nHibernate — полнофункциональный ORM-фреймворк. Оба фреймворка распространяются с открытым кодом и развиваются по инициативе сообщества, но на момент написания книги только в Dapper присутствовала поддержка .NET Core. Главное отличие между микро-ORM и полноценным ORM-фреймворком (таким, как Entity Framework) заключается в том, что микро-ORM обычно требует вручную вводить выполняемые команды SQL, а ORM генерируют большинство команд SQL за вас на основании имеющейся информации о сущностях и базе данных. Выбор между ORM и микро-ORM в конечном итоге зависит от команды и тех задач, которые вы пытаетесь решить.

Команда Alpine Ski House решила использовать EF Core, потому что участники были знакомы с предыдущими версиями Entity Framework и им нравилось направление, в котором группа ASP.NET развивала EF Core: система становилась относительно легкой и проще расширялась по сравнению с предыдущими версиями EF. Другим преимуществом была поддержка LINQ, используемая для эффективного построения запросов к модели за счет выражения содержательных запросов в коде C#. EF Core также предоставляет простой механизм использования SQL в случае необходимости — по соображениям производительности или просто для удобства.

ВСЕ ЭТО SYSTEM.DATA

Стоит заметить, что независимо от выбора ORM для вашего приложения все они строятся на базе System.Data. System.Data API, которые были частью .NET с версии 1, предоставляют абстракцию для взаимодействия с базами данных в форме IDbConnection, IDbCommand, IDataReader и IDbTransaction. Практически все ORM, написанные в истории .NET, были построены на базе этих четырех интерфейсов.

На самом деле это довольно-таки замечательный факт, если вспомнить, как давно существует .NET; по сути это свидетельство высокого качества проектирования, выполненного при создании .NET.

Разные провайдеры баз данных могут предоставлять собственные реализации основных интерфейсов System.Data. Microsoft предоставляет готовую реализацию для SQL Server в System.Data.SqlClient; также доступны многие другие реализации, включая такие распространенные, как MySQL, Oracle, PostgreSQL и SQLite. Команда Alpine Ski House использует SQL Server из-за превосходной кроссплатформенной поддержки в .NET Core в сочетании с качественным техническим обеспечением.

Основы Entity Framework

При использовании Entity Framework в качестве прослойки для работы с данными выражение модели предметной области осуществляется через POCO-объекты (Plain Old CLR Objects). Например, сущность предметной области «карта» может быть просто выражена как класс `SkiCard`. Entity Framework берет на себя все хлопоты по сопоставлению свойств класса со столбцами конкретной таблицы базы данных. Entity Framework управляет отображением и любыми взаимодействиями с базой данных через экземпляр класса, наследующий от `DbContext`.

Прежде чем браться за глубокое изучение предметной области Alpine Ski House, рассмотрим упрощенный пример. Представьте очень простую базу данных SQL Server, в которой список всех курортов, находящихся под управлением Alpine Ski House, хранится в таблице `Resorts` (табл. 10.1).

Таблица 10.1. Таблица `Resorts`

Resorts	
Id	int
Name	nvarchar(255)
OperatingSince	date

Приложение моделирует запись `Resort` при помощи POCO-объекта с именем `Resort`. Такие классы называются классами сущностей (entity classes).

```
public class Resort
{
    public int Id { get; set; }
    public string Name { get; set; }
    public DateTime OperatingSince {get; set;}
}
```

Затем необходимо добавить ссылку на пакет `Microsoft.EntityFrameworkCore.SqlServer`, который извлекает все необходимые компоненты EF Core. Но пока EF ничего не знает о сущности `Resort`. Чтобы сообщить EF эту информацию, создайте класс `DbContext`, который содержит объект `DbSet` для `Resorts`.

```
public class ResortContext : DbContext
{
    public DbSet<Resort> Resorts {get; set;}

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        //Не указывайте здесь строки подключений при создании реальных
        //приложений
        //В приложениях ASP.NET они настраиваются с использованием
        //внедрения зависимостей.
        //См. главы 11 и 13
        optionsBuilder.UseSqlServer("Server=(localdb)\\MSSQLLocalDB;Database=ResortDBs;
            Integrated
            Security=true");
    }
}
```

Вот и все, что необходимо для начала использования Entity Framework. Соглашения EF обеспечат отображение класса `Resort` на таблицу `Resorts` в базе данных. Все эти соглашения могут переопределяться по мере необходимости, но для начала рассмотрим базовый API `DbContext`, предоставляющий полезную абстракцию базы данных. Чтобы обратиться к данным в базе данных, начните с создания экземпляра `ResortContext`.

```
var context = new ResortContext();
//Использование контекста
context.Dispose();
```

Важно вызвать метод `Dispose` после завершения работы с экземпляром `DbContext`. Вызов метода `Dispose` гарантирует, что все подключения к базе данных будут возвращены в пул подключений. Если подключения не будут возвращаться в пул, возникает риск истощения пула подключений, что в конечном итоге приведет к тайм-аутам. При ручном создании контекста рекомендуется использовать паттерн `using`, чтобы гарантировать вызов метода `Dispose` при завершении блока кода.

```
using(var context = new ResortContext())
{
    //Использование контекста
}
```

В приложении ASP.NET используйте встроенный фреймворк внедрения зависимостей для создания и уничтожения контекстов. Эта тема более подробно рассматривается в главе 14 «Внедрение зависимостей».

Запрос на получение одной записи

Для получения одной записи по первичному ключу используется метод `DbSet.Find`:

```
var resort = context.Resorts.Find(12);
```

Если запрос для получения одной записи основан на другом критерии вместо первичного ключа, используйте метод `First` и укажите лямбда-выражение, возвращающее `true` для подходящей записи:

```
var resort = context.Resorts.First(f => f.Name == "Alpine Ski House");  
Resulting Query:  
SELECT TOP(1) [f].[Id], [f].[Name], [f].[OperatingSince]  
FROM [Resorts] AS [f]  
WHERE [f].[Name] = N'Alpine Ski House'
```

Если не найдено ни одного совпадения, метод `First` выдает исключение. Также можно воспользоваться методом `FirstOrDefault`, который возвращает `Null`, если не найдено ни одного совпадения.

Другое возможное решение — использовать методы `Single` и `SingleOrDefault`. Эти методы работают практически так же, как методы `First` и `FirstOrDefault`, но они выдают исключение при обнаружении нескольких подходящих записей.

```
var resort = context.Resorts.Single(f => f.Name == "Alpine Ski House");
```

```
Resulting Query:  
SELECT TOP(2) [f].[Id], [f].[Name], [f].[OperatingSince]  
FROM [Resorts] AS [f]  
WHERE [f].[Name] = N'Alpine Ski House'
```

Обратите внимание: EF генерирует для `Single` запрос `SELECT TOP(2)` — вместо `SELECT TOP(1)`, как для запроса `First`. Запрос `SELECT TOP(2)` необходим для того, чтобы при обнаружении нескольких записей фреймворк EF мог выдать исключение. Используйте `Single`, если вы ожидаете найти только одну подходящую запись, потому что этот метод более четко выражает смысл запроса и выдает четко выраженную ошибку, если ожидания не оправдываются.

Запрос на получение нескольких записей

Во многих случаях из базы данных требуется запросить несколько записей, удовлетворяющих некоторому критерию. Задача решается вызовом метода `Where` объекта `DbSet`; метод возвращает все записи, соответствующие заданному лямбда-выражению.

```
var skiResorts = context.Resorts.Where(f => f.Name.Contains("Ski")).ToList();
```

```
Resulting Query:  
SELECT [f].[Id], [f].[Name], [f].[OperatingSince]
```



```
FROM [Resorts] AS [f]
WHERE [f].[Name] LIKE (N'%' + N'Ski') + N'%'
```

EF не обращается с вызовом к базе данных непосредственно в момент вызова `Where`. Вместо этого возвращается объект `IQueryable<T>`, где `T` — тип запрашиваемой сущности. Фактическое выполнение запроса откладывается до момента вызова для `IQueryable<T>` метода, требующего выполнения запроса. В данном случае вызов метода `ToList` форсирует немедленное выполнение. Отложенное выполнение запросов — важная особенность EF, которая позволяет объединять вызовы методов в цепочку и строить сложные запросы. Это пример паттерна отложенного выполнения, при котором создание объекта или вычисление значения откладывается до того момента, когда оно будет впервые затребовано.

Сохранение данных

`DbContext` также предоставляет механизм для отслеживания изменений в сущностях и сохранения этих изменений в базе данных. Рассмотрим основы сохранения изменений.

Вставка данных реализуется добавлением новой сущности в `DbSet` с последующим вызовом метода `SaveChanges` объекта `DbContext`:

```
var resort = new Resort
{
    Name = "Alpine Ski House 2",
    OperatingSince = new DateTime(2001, 12, 31)
};
context.Resorts.Add(resort);
context.SaveChanges();

Resulting Query:
INSERT INTO [Resorts] ([Name], [OperatingSince])
VALUES (@p0, @p1);
```

Перед вызовом `SaveChanges` в `DbSet` можно добавить несколько сущностей. Entity Framework отслеживает все новые сущности. При вызове `SaveChanges` EF отправляет базе данных команды вставки. В этот момент создается транзакция, а все команды вставки выполняются в этой транзакции. Если хотя бы одна команда вставки завершится неудачей, то происходит откат всех команд.

Отслеживание изменений

Система отслеживания изменений `DbContext` также следит за изменениями в имеющихся сущностях и выдает команды `Update` при вызове `SaveChanges`:

```
var resort= context.Resorts.Find(12);
resort.Name = "New Name";
context.SaveChanges();
```

Resulting Query:

```
UPDATE [Resorts] SET [Name] = @p1  
WHERE [Id] = @p0;
```

Обновляется только столбец `[Name]`. Другие столбцы не обновляются, потому что система отслеживания изменений знает, что эти столбцы не изменились. Отслеживание изменений связано с определенными дополнительными затратами. Если сущности загружаются только для чтения, возможно, будет полезно отключить отслеживание изменений. Это можно сделать в каждом отдельном случае с использованием метода расширения `AsNoTracking()`.

```
var lastYear = new DateTime(2015,1,1);  
var newResorts = context.Resorts.AsNoTracking().Where(f => f.OperatingSince >  
    lastYear);
```

Метод `DbSet.Remove` используется для удаления сущностей.

```
var resort= context.Resorts.Find(12);  
context.Resorts.Remove(resort);  
context.SaveChanges();
```

Resulting Query:

```
DELETE FROM [Resorts]  
WHERE [Id] = @p0;
```

Использование миграции для создания и обновления баз данных

До сих пор наш простой пример предполагал, что в системе уже существует база данных, соответствующая нашим сущностям. Во многих случаях, включая приложение `Alpine Ski House`, вам придется работать с совершенно новой базой данных. Разработчикам нужен простой механизм создания локальной тестовой базы данных. Также требуется определять и выполнять миграционные сценарии при внесении изменений в базу данных. EF предоставляет инструментарий командной строки для автоматизации этих задач. Инструментарий командной строки включается через NuGet посредством добавления зависимости для пакета `Microsoft.EntityFrameworkCore.Tools` и выполнением команды `dotnet ef`.

Управление зависимостями более подробно рассматривается в главе 16 «Управление зависимостями».

Создание новой базы данных

Новая база данных создается командой `dotnet ef database update`:

```
> dotnet ef database update
```

Команда компилирует приложение и подключается к базе данных с использованием строки подключения, указанной в конфигурации локальной среды разра-

ботки. Конфигурация более подробно рассматривается в главе 12 «Конфигурация и журналирование».

Если указанная база данных не существует, создается новая база данных. Если база данных существует, применяются все миграционные сценарии, необходимые для обновления базы данных и приведения ее в соответствие с классами предметной области в приложении.

Вы также можете явно задать имя класса `DbContext`, который должен использоваться командой.

```
> dotnet ef database update -c AlpineSkiHouse.Data.ResortContext
```

Параметр `-c` (или `--context`) может передаваться любой из команд `dotnet ef`, и он необходим в том случае, если в приложении используется более одного класса `DbContext`. Как видно из приложения Alpine Ski House, этот параметр широко применяется на практике.

В результате выполнения команды `dotnet ef database update` для `ResortContext` создается новая база данных, которая содержит только таблицу `__EFMigrationsHistory` (рис. 10.1).



Рис. 10.1. Новая база данных создается командой `dotnet ef database update`

Вопреки ожиданиям, база данных не содержит таблицы `Resorts`, потому что вы еще не определили миграции для `ResortContext`.

Добавление миграций

Миграции Entity Framework используются для обновления (повышения или снижения версии) базы данных. Миграция определяется в виде класса C#, наследу-

ющего от класса `Migration`. Класс содержит два метода: `Up()` для повышающего обновления версии, определяемой миграцией, и `Down()` для возврата к предыдущей версии.

Определить миграции вручную было бы крайне утомительно, поэтому Entity Framework поддерживает команду `dotnet ef migrations add`, которая берет на себя большую часть рутинной работы. В случае нового класса `DbContext` необходима новая миграция для создания таблиц вашей модели.

```
dotnet ef migrations add CreateInitialSchema
```

Эта команда создает новый класс с именем `CreateInitialSchema` в папке `Migrations` рядом с заданным классом `DbContext`.

```
public partial class CreateInitialSchema : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "Resorts",
            columns: table => new
            {
                Id = table.Column<int>(nullable: false)
                    .Annotation("SqlServer:ValueGenerationStrategy",
                        SqlServerValueGenerationStrategy.IdentityColumn),
                Name = table.Column<string>(nullable: true),
                OperatingSince = table.Column<DateTime>(nullable: false)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_Resorts", x => x.Id);
            });
    }

    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable(
            name: "Resorts");
    }
}
```

В данном случае метод `Up` добавляет таблицу `Resorts`, в которой каждому свойству класса `Resort` соответствует свой столбец. Метод `Down` удаляет таблицу `Resorts`. В локальной среде разработки `CreateInitialSchema` считается незавершенной (pending) миграцией, потому что она еще не была применена к локальной базе данных. Команда `dotnet ef database update` применяет незавершенное обновление и создает таблицу `Resorts` (рис. 10.2).

```
dotnet ef database update
```

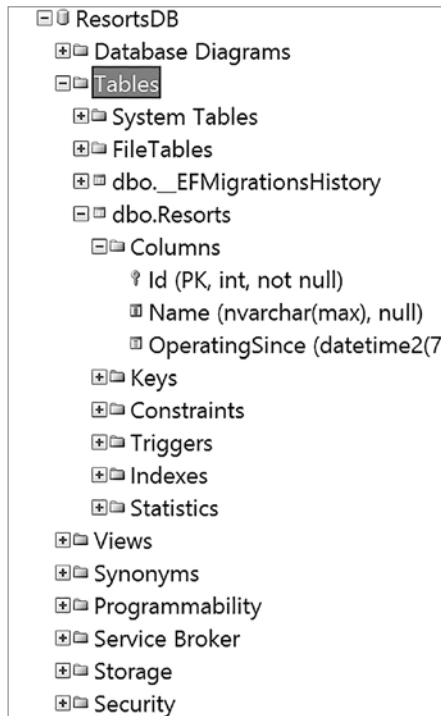


Рис. 10.2. База данных ResortsDB с применением незавершенных изменений

ПРИМЕЧАНИЕ

Возможно, вас удивит использование типа данных `nvarchar(max)` для столбца `Name`. Без передачи дополнительных параметров Entity Framework отображает строковые типы .NET на `nvarchar(max)`. Аннотации данных и конфигурация модели рассматриваются позже в этой главе.

При внесении изменений в класс предметной области или при добавлении новых классов в папке `Migrations` должна определяться новая миграция. Типичный пример такого рода — добавление нового свойства в класс предметной области.

```
public class Resort
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string MailingAddress { get; set; } // Добавлено
    public DateTime OperatingSince { get; set; }
}
```

После добавления свойства происходит рассинхронизация модели предметной области и базы данных, и для согласования базы данных с моделью требуется

миграция. Начните с выполнения команды `dotnet ef migrations add` с указанием содержательного имени миграции.

```
> dotnet ef migrations add AddedMailingAddress
```

Entity Framework компилирует ваше приложение и находит любые незавершенные изменения в классах сущностей; в папку Migrations добавляется новый класс.

```
public partial class AddedMailingAddress : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.AddColumn<string>(
            name: "MailingAddress",
            table: "Resorts",
            nullable: true);
    }

    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropColumn(
            name: "MailingAddress",
            table: "Resorts");
    }
}
```

В данном случае метод `Up` добавляет новый столбец `MailingAddress`, а метод `Down` удаляет столбец `MailingAddress`. Команда `dotnet ef database update` применяет незавершенную миграцию.

Инструментарий миграций EF неплохо справляется с генерацией кода для изменения структуры базы данных, но иногда также приходится вносить изменения в данные командами `UPDATE`, `INSERT` или `DELETE`. Изменения в данных могут быть реализованы передачей любой синтаксически правильной команды SQL методу `migrationBuilder.Sql` в методе `Up` или `Down`. Возможно даже добавление миграции без изменения модели. Команда `dotnet ef migrations add` без изменения модели приводит к пустой миграции. Ниже приведен пример миграции, которая просто добавляет строки в таблицу `Resorts`.

```
public partial class AddResortsAndLocations : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.Sql(
            @"INSERT INTO Resorts(Name)
              VALUES('Alpine Ski House'),
                ('Nearby Resort'),
                ('Competing Resort')");
    }
}
```

```
protected override void Down(MigrationBuilder migrationBuilder)
{
    migrationBuilder.Sql(
        @"DELETE Resorts WHERE Name IN
        ('Alpine Ski House',
        'Nearby Resort',
        'Competing Resort')");
}
}
```

ПРИМЕЧАНИЕ

Как правило, классы сущностей и классы DbContext не содержат логики, относящейся к провайдеру базы данных. С другой стороны, миграции привязаны к провайдеру, настроенному на момент создания миграции. Следовательно, не стоит рассчитывать на то, что вы создадите миграцию для SQL Server, а другой разработчик на другом компьютере сможет применить эту миграцию для SQL Lite. Если вы решили поддерживать несколько провайдеров баз данных, попробуйте организовать управление миграционными сценариями вручную, за пределами Entity Framework.

Создание сценариев обновления для рабочих серверов

Команда `dotnet ef database update` хорошо работает для машин разработчиков, но не всегда подходит для развертывания на рабочем сервере базы данных. В тех случаях, когда ваш билд-сервер не может применять миграции напрямую, используйте команду `dotnet ef migrations script` для генерации сценария базы данных, который может использоваться для применения незавершенных миграций к базе данных.

```
C:\EFBasics>dotnet ef migrations script
Project EFBasics (.NETCoreApp,Version=v1.0) was previously compiled. Skipping
    compilation.
IF OBJECT_ID(N'__EFMigrationsHistory') IS NULL
BEGIN
    CREATE TABLE [__EFMigrationsHistory] (
        [MigrationId] nvarchar(150) NOT NULL,
        [ProductVersion] nvarchar(32) NOT NULL,
        CONSTRAINT [PK__EFMigrationsHistory] PRIMARY KEY ([MigrationId])
    );
END;
GO
CREATE TABLE [Resorts] (
    [Id] int NOT NULL IDENTITY,
    [Name] nvarchar(max),
    [OperatingSince] datetime2 NOT NULL,
    CONSTRAINT [PK_Resorts] PRIMARY KEY ([Id])
);
GO
INSERT INTO [__EFMigrationsHistory] ([MigrationId], [ProductVersion])
```

```
VALUES (N'20160627025825_CreateInitialSchema', N'1.0.0-rc2-20901');
GO
ALTER TABLE [Resorts] ADD [MailingAddress] nvarchar(max);
GO
INSERT INTO [__EFMigrationsHistory] ([MigrationId], [ProductVersion])
VALUES (N'20160627030609_AddedMailingAddress', N'1.0.0-rc2-20901');
GO
```

С параметрами по умолчанию эта команда выводит все миграции для заданного класса `DbContext`. Параметры `From` и `To` могут использоваться для генерации сценариев миграции между двумя конкретными версиями.

```
>dotnet ef migrations script 0 InitialResortContext
Project AlpineSkiHouse.Web (.NETCoreApp,Version=v1.0) was previously compiled.
Skipping
compilation.
IF OBJECT_ID(N'__EFMigrationsHistory') IS NULL
BEGIN
    CREATE TABLE [__EFMigrationsHistory] (
        [MigrationId] nvarchar(150) NOT NULL,
        [ProductVersion] nvarchar(32) NOT NULL,
        CONSTRAINT [PK__EFMigrationsHistory] PRIMARY KEY ([MigrationId])
    );
END;
GO
CREATE TABLE [Resorts] (
    [Id] int NOT NULL IDENTITY,
    [Name] nvarchar(max),
    CONSTRAINT [PK_Resorts] PRIMARY KEY ([Id])
);
GO
CREATE TABLE [Locations] (
    [Id] int NOT NULL IDENTITY,
    [Altitude] decimal(18, 2),
    [Latitude] decimal(18, 2),
    [Longitude] decimal(18, 2),
    [Name] nvarchar(max),
    [ResortId] int NOT NULL,
    CONSTRAINT [PK_Locations] PRIMARY KEY ([Id]),
    CONSTRAINT [FK_Locations_Resorts_ResortId] FOREIGN KEY ([ResortId]) REFERENCES
        [Resorts]
        ([Id]) ON DELETE CASCADE
);
GO
CREATE INDEX [IX_Locations_ResortId] ON [Locations] ([ResortId]);
GO
INSERT INTO [__EFMigrationsHistory] ([MigrationId], [ProductVersion])
VALUES (N'20160625231005_InitialResortContext', N'1.0.0-rtm-21431');
GO
```


Теперь, после рассмотрения основных компонентов Entity Framework, посмотрим, как использовать их для реализации веб-приложения Alpine Ski House.

ApplicationDbContext

Создавая проект `AlpineSkiHouse.Web` в Visual Studio, мы выбрали шаблон `Web Application` в режиме `Individual User Accounts`. Выбор этого режима создает в приложении контекст с именем `ApplicationDbContext` — `DbContext` для сущности `ApplicationUser`. Рассмотрим сущность и контекст, добавленный в проект:

```
Models\ApplicationUser.cs
namespace AlpineSkiHouse.Models
{
    // Добавьте профильные данные пользователей приложения,
    // включая свойства в класс ApplicationUser
    public class ApplicationUser : IdentityUser
    {
    }
}
```

Класс `ApplicationUser` наследует от класса `IdentityUser` основные свойства пользователя — такие, как `EmailAddress` (адрес электронной почты) и `Id` (уникальный идентификатор). Как сказано в комментарии, класс можно расширить с добавлением собственных свойств. Именно это и понадобится для текущего спринта, но пока рассмотрим файл `ApplicationDbContext`.

```
Data\ApplicationDbContext.cs
namespace AlpineSkiHouse.Data
{
    public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
    {
        public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
            : base(options)
        {
        }

        protected override void OnModelCreating(ModelBuilder builder)
        {
            base.OnModelCreating(builder);
        }
    }
}
```

Класс `ApplicationDbContext`, как и `ApplicationUser`, наследует свойства от базового класса: `IdentityDbContext`. В случае `ApplicationDbContext` эти свойства представляют собой объекты `DbSet` для сущностей, связанных с ASP.NET Identity: `Users`, `Roles`, `Claims` и `Tokens`. Сущности ASP.NET Identity более по-

дробно рассматриваются в главе 13 «Идентификация, безопасность и управление правами».

Вместо принудительного описания конфигурации в методе `OnConfiguration` параметры конфигурации передаются конструктору. Это позволяет вам задать конфигурацию в процессе запуска приложения и использовать встроенные средства внедрения зависимостей для включения ее в `DbContext`. Используйте такую сигнатуру конструктора для всех классов `DbContext`, которые вы определяете в своем приложении. Использование этого механизма конфигурации упрощает тестирование классов `DbContext`; эта тема более подробно рассматривается в главе 19 «Тестирование».

Когда вы начинаете добавлять функциональность в приложение Alpine Ski House, у вас может возникнуть заманчивая идея разместить все сущности в классе `ApplicationDbContext` — более того, поначалу это даже удобно. Однако вам будет намного проще заниматься сопровождением приложения, если вы создадите отдельные классы `DbContext` для всех ограниченных контекстов, о которых говорилось в главе 4 «Структура проекта». `ApplicationDbContext` соответствует ограниченному контексту входа, но имя порождает путаницу: оно создает впечатление, что это единый контекст `DbContext` для всего приложения, поэтому `ApplicationDbContext` следует переименовать в `ApplicationUserContext`.

```
public class ApplicationUserContext : IdentityDbContext<ApplicationUser>
{
    public ApplicationUserContext(DbContextOptions<ApplicationUserContext> options)
        : base(options)
    {
    }

    protected override void OnModelCreating(ModelBuilder builder)
    {
        base.OnModelCreating(builder);
    }
}
```

Расширение `ApplicationUserContext`

Итак, в вашей предметной области теперь существуют четкие границы, и вы можете добавить в класс `ApplicationUser` дополнительную информацию о пользователе. Милдред, специалист по продажам и маркетингу, хотела бы хранить имена для рассылки персональных сообщений. Кроме того, она требует хранить контактные данные пользователей (например, адреса электронной почты и телефонные номера), чтобы оповещать пользователей об операциях с их учетными записями или отправлять сводки после дня, проведенного на одном из курортов фирмы.

Свойства `EmailAddress` и `PhoneNumber` уже являются частью базового класса `IdentityUser`, но свойства `FirstName` и `LastName` необходимо добавить:

```
public class ApplicationUser : IdentityUser
{
    [MaxLength(70)]
    [Required]
    public string FirstName { get; set; }

    [MaxLength(70)]
    public string LastName { get; set; }
}
```

Аннотации данных используются здесь для определения обязательных свойств, а также для определения максимальной длины строковых свойств. Эти атрибуты используются для проверки состояния сущности перед ее сохранением; кроме того, Entity Framework использует их при генерации типов данных столбцов для базы данных. Так как мы задали `MaxLength` значение 70 для полей `FirstName` и `LastName`, столбцу назначается тип `nvarchar(70)` вместо `nvarchar(max)`. Поле `LastName` помечается как необязательное, потому что некоторые пользователи могут указать только имя ввиду отсутствия фамилии. И хотя в североамериканских культурах такая ситуация с именами встречается редко, нельзя запрещать клиенту доступ на курорт из-за своенравно установленных ограничений на фамилии. Пусть Шер или Мадонна спокойно приезжают на курорты Alpine Ski House на свою ежегодную лыжную прогулку.

После внесения изменений в класс сущности необходимо создать миграцию для добавления этих столбцов в базу данных.

```
dotnet ef migrations add AddNamePropertiesToApplicationUser
```

```
public partial class AddNamePropertiesToApplicationUser : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.AddColumn<string>(
            name: "FirstName",
            table: "AspNetUsers",
            nullable: false,
            defaultValue: "");

        migrationBuilder.AddColumn<string>(
            name: "LastName",
            table: "AspNetUsers",
            nullable: true);
    }

    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropColumn(
            name: "FirstName",
```

```
        table: "AspNetUsers");  
  
        migrationBuilder.DropColumn(  
            name: "LastName",  
            table: "AspNetUsers");  
    }  
}
```

Чтобы применить эту миграцию, выполните команду `dotnet ef database update`.

Контекст для карты

Карты — одна из важных концепций проекта Parsley. Карта представляет собой физическую карточку, которую клиент берет с собой при выходе на склон; эта карточка распознается сканерами в лифтах и других местах. Карта представляет клиента в системе. В исходной модели клиент и карта моделируются как две разные сущности. Поскольку карта однозначно связывается с клиентом и один не может существовать без другого, мы решили объединить их в одну сущность `SkiCard`. После входа пользователь получает возможность создать одну или несколько карт для себя и членов своей семьи.

Сначала создайте новый класс `DbContext` с именем `SkiCardContext`.

```
public class SkiCardContext : DbContext  
{  
    public SkiCardContext(DbContextOptions<SkiCardContext> options) : base(options)  
    {  
    }  
  
    public DbSet<SkiCard> SkiCards { get; set; }  
}
```

Контекст с одним или двумя типами сущностей может показаться странным, но в стремлении к минимизации нет ничего плохого. Затраты на поддержание множественных контекстов в системе минимальны, к тому же вы всегда можете провести рефакторинг позднее.

А пока настройте `SkiCardContext` для хранения данных в той же базе данных с `ApplicationUserContext`. Конфигурация строки подключения определяется в методе `ConfigureServices` класса `Startup`.

```
services.AddDbContext<SkiCardContext>(options =>  
    options.UseSqlServer(Configuration.GetConnectionString(  
        "DefaultConnection")));
```

Класс `SkiCard` содержит уникальный идентификатор, дату и время создания карты, а также основную информацию о клиенте, которого мы будем называть «держателем карты».

```
public class SkiCard
{
    public int Id { get; set; }

    /// <summary>
    /// Идентификатор пользователя ApplicationUser - владельца карты.
    /// </summary>
    public string ApplicationUserId { get; set; }

    /// <summary>
    /// Дата создания карты.
    /// </summary>
    public DateTime CreatedOn { get; set; }

    [MaxLength(70)]
    [Required]
    public string CardHolderFirstName { get; set; }

    [MaxLength(70)]
    public string CardHolderLastName { get; set; }

    public DateTime CardHolderBirthDate { get; set; }

    [Phone]
    public string CardHolderPhoneNumber { get; set; }
}
```

Команда Alpine Ski House согласилась с тем, что в будущем сущность `SkiCard` нужно будет расширять, но сейчас она станет хорошей отправной точкой для достижения цели спринта 2: предоставить возможность выполнить вход и создать абонементы для себя и членов семьи. А теперь, после создания исходной версии ограниченного контекста для карты, мы создадим исходную миграцию командой `dotnet ef migrations command` и обновим базу данных локальной разработки командой `dotnet ef database update`.

```
dotnet ef migrations add InitialSkiCardContext -c AlpineSkiHouse.Data.
    SkiCardContext
dotnet ef database update -c AlpineSkiHouse.Data.SkiCardContext
```

Отношения, выходящие за границы контекстов

Вероятно, любой специалист с опытом проектирования реляционных баз данных будет в ужасе от того, что мы здесь делаем. Определяя границы вокруг классов `DbContext`, мы разрываем связи между таблицами, которые обычно соединяются через внешний ключ. Как обеспечить ссылочную ценность без внешних ключей? Для примера возьмем карты. Таблица `SkiCard` содержит столбец с именем `ApplicationUserId`, в котором хранится ссылка на первичный ключ объекта `ApplicationUser`, создавшего этот экземпляр `SkiCard`. Если бы эти сущности нахо-

дились в одном ограниченном контексте, мы бы явно смоделировали отношения между `ApplicationUser` и `SkiCard` при помощи навигационных свойств. С навигационными свойствами EF свяжет две таблицы внешним ключом, а ссылочная целостность данных будет обеспечиваться базой данных.

В данных условиях отношение не определяется явным образом. Команда `Alpine Ski House` обсудила плюсы и минусы такого решения. Одна из основных проблем связана с тем, что произойдет, если пользователь захочет удалить свою учетную запись. Без явных отношений между `ApplicationUser` и `SkiCard` появятся «потерянные» записи `SkiCard`, не имеющие владельца. Команда рассмотрела пару возможных решений. Первый вариант — вручную определить внешний ключ с использованием миграции.

```
dotnet ef migrations add AddSkiCardApplicationUserForeignKey -c AlpineSkiHouse.  
Data.  
SkiCardContext
```

Так как модель не изменяется, эта команда создает пустую миграцию, в которую можно вручную добавить внешний ключ.

```
public partial class AddSkiCardApplicationUserForeignKey : Migration  
{  
    protected override void Up(MigrationBuilder migrationBuilder)  
    {  
        migrationBuilder.AddForeignKey("FK_SkiCards_ApplicationUser_  
                                        ApplicationUserID",  
                                        "SkiCards", "ApplicationUserID",  
                                        "AspNetUsers", principalColumn: "Id",  
                                        onDelete: ReferentialAction.Cascade);  
    }  
  
    protected override void Down(MigrationBuilder migrationBuilder)  
    {  
        migrationBuilder.DropForeignKey("FK_SkiCards_ApplicationUser_  
                                        ApplicationUserID",  
                                        "SkiCards");  
    }  
}
```

Это гарантирует, что на уровне реляционной базы данных никогда не появятся потерянные записи `SkiCard`, но этот метод может породить другие проблемы. Даже если пользователь захочет удалить свою учетную запись, сторона бизнеса не всегда хочет, чтобы запись `SkiCard` пропала из базы. Ей по-прежнему нужно знать, что карта существует, а также то, какие абонементы были использованы для этой карты. Удаление записей карт может привести к тому, что отчеты на основании истории будут содержать неточную информацию. Вы можете изменить значение `onDelete` на `ReferentialAction.SetDefault`, но команда быстро осознает,

что обсуждаемая функциональность выходит за рамки текущего спринта. Впрочем, пользователи все равно не имеют возможности для удаления своих учетных записей, поэтому не стоит тратить время на попытки угадать будущие требования. А пока было принято решение об удалении миграции первичного ключа, а решения по этой функциональности откладываются на будущее.

Удаление учетной записи пользователя — нетривиальная операция, последствия которой распространяются по нескольким ограниченным контекстам. Вместо того чтобы пытаться реализовать всю эту логику через внешние ключи, лучше обработать ее при помощи событий прослушивания — она станет более понятной и простой для расширения. При удалении `ApplicationUser` ограниченный контекст пользователя приложения инициирует событие `ApplicationUserDeleted`. Любые действия, которые должны происходить в результате события `ApplicationUserDeleted`, реализуются в виде обработчиков событий в других ограниченных контекстах. Такое решение считается предпочтительным; используйте его, если вам нужна эта функциональность.

Присоединение контроллера

Теперь, когда контекст стал доступным, необходимо реализовать контроллер для сущности `SkiCard`. Контроллер предоставляет конечные точки для просмотра списка лыжных карт, создания новой карты и правки информации существующей карты.

Этому контроллеру необходим доступ к классам `SkiCardContext` и `UserManager<ApplicationUser>`, которые передаются конструктору. Зависимости разрешаются встроенным фреймворком внедрения зависимостей (см. главу 14).

```
[Authorize]
public class SkiCardController : Controller
{
    private readonly SkiCardContext _skiCardContext;
    private readonly UserManager<ApplicationUser> _userManager;

    public SkiCardController(SkiCardContext skiCardContext,
        UserManager<ApplicationUser> userManager)
    {
        _skiCardContext = skiCardContext;
        _userManager = userManager;
    }
}
```

Добавление атрибута `[Authorize]` в `SkiCardController` гарантирует, что доступ к методам действий `SkiCardController` будет предоставлен только тем пользователям, которые ввели свои входные данные. Аутентификация и авторизация более подробно рассматриваются в главе 13.

Метод действия Index

Начнем с реализации метода действия `Index`. Метод действия не получает аргументов и отвечает за получение всех карт текущего пользователя, преобразование каждой карты в экземпляр `SkiCardListViewModel` и передачу списка моделей представления объекту `View` для визуализации в формате HTML. В соответствии со стандартными конфигурациями маршрутизации MVC URL-адрес этой конечной точки имеет вид `/SkiCard`.

Чтобы получить список карт для текущего пользователя, сначала получите `userId` текущего пользователя из экземпляра `userManager`.

```
var userId = _userManager.GetUserId(User);
```

Затем все карты, у которых `ApplicationUserId` совпадает с `userId`, запрашиваются через `DbSet _skiCardContext.SkiCards`.

```
_skiCardContext.SkiCards.Where(s => s.ApplicationUserId == userId)
```

Вместо того чтобы выполнять запрос и перебирать результаты для создания экземпляра `SkiCardListViewModel` для каждого объекта `SkiCard`, вы можете определить проекцию LINQ с использованием метода-проекции `Select` для получения из базы данных только свойств, необходимых для создания `SkiCardListViewModels`.

```
_skiCardContext.SkiCards
    .Where(s => s.ApplicationUserId == userId)
    .Select(s => new SkiCardListViewModel
    {
        Id = s.Id,
        CardHolderName = s.CardHolderFirstName + " " + s.CardHolderLastName
    });
```

Метод-проекция `Select`, использующий механизм отложенного выполнения запросов, о котором говорилось ранее, указывает, что вместо возвращения экземпляров класса `SkiCard` следует возвращать экземпляры `SkiCardListViewModel`, использующие значения свойств класса `SkiCard`.

Entity Framework хватает интеллекта для того, чтобы сгенерировать запрос, который возвращает только данные, необходимые для заполнения свойств `SkiCardListViewModel`.

```
SELECT [s].[Id], ([s].[CardHolderFirstName] + N' ') + [s].[CardHolderLastName]
FROM [SkiCards] AS [s]
WHERE [s].[ApplicationUserId] = @__userId_0
```

Вот как выглядит полный код метода действия `Index`, который передает представлению список моделей представления. Определение представлений с использованием `Razor` более подробно рассматривается в главе 11 «Представления `Razor`».


```
// GET: SkiCard
public async Task<ActionResult> Index()
{
    var userId = _userManager.GetUserId(User);
    var skiCardsViewModels = await _skiCardContext.SkiCards
        .Where(s => s.ApplicationUserId == userId)
        .Select(s => new SkiCardListViewModel
        {
            Id = s.Id,
            CardHolderName = s.CardHolderFirstName + " " + s.CardHolderLastName
        })
        .ToListAsync();

    return View(skiCardsViewModels);
}
```

ASYNС/AWAIT

В большинстве методов действий, связанных с выдачей запросов к базе данных или работой с неким внешним ресурсом, используется конструкция `async/await`. Дополнительную информацию об `async` и `await` в приложениях ASP.NET можно найти в отличном видео на Channel 9: <https://channel9.msdn.com/Events/aspConf/aspConf/Async-in-ASP-NET>.

Методы действий Create

Перейдем к реализации методов действий **Create**, которые используются для создания новой карты. Контроллер **SkiCard** содержит два метода действий **Create**, обоим соответствует URL-адрес *SkiCard/Create*. Первый метод действия не получает аргументов, вызывается для HTTP-команды GET и возвращает представление с формой, заполняемой пользователем. Второй метод действия получает аргумент **CreateSkiCardViewModel** и вызывается для HTTP-команды POST. Метод действия POST отвечает за проверку значений модели представления и создание новой сущности **SkiCard**. После успешного создания сущности **SkiCard** он перенаправляет пользователя на метод действия **Index**. Эта схема поведения обычно называется «POST/Redirect/GET».

Типичный метод GET для действия создания не содержит большого объема логики. В данном случае объект **CreateSkiCardViewModel** должен быть заполнен некоторыми значениями по умолчанию. В частности, можно считать, что номер телефона (**PhoneNumber**) держателя карты совпадает с номером телефона текущего пользователя.

Также можно предположить, что если у пользователя еще нет ни одной существующей карты, то он создает карту для себя. Свойства **FirstName** и **LastName** можно заранее заполнить именем и фамилией текущего пользователя. Конечно, у поль-

зователя будет возможность изменить эти значения на форме, если заранее заполненные данные окажутся неправильными.

```
// GET: SkiCard/Create
public async Task<ActionResult> Create()
{
    var userId = _userManager.GetUserId(User);
    var currentUser = await _userManager.FindByIdAsync(userId);
    var viewModel = new CreateSkiCardViewModel
    {
        CardHolderPhoneNumber = currentUser.PhoneNumber
    };

    // Если это первая карта пользователя, она автоматически заполняется
    // свойствами имен, потому что карта с большой вероятностью предназначена
    // для этого пользователя. В противном случае предполагается, что карта
    // предназначена для члена семьи, а свойства имен остаются пустыми.
    var hasExistingSkiCards = _skiCardContext.SkiCards.Any(s =>
        s.ApplicationUserId == userId);

    if (!hasExistingSkiCards)
    {
        viewModel.CardHolderFirstName = currentUser.FirstName;
        viewModel.CardHolderLastName = currentUser.LastName;
    }

    return View(viewModel);
}
```

На основании информации, полученной с использованием метода `Any`, EF генерирует эффективный запрос для проверки наличия существующих карт для заданного пользователя.

```
SELECT CASE
    WHEN EXISTS (
        SELECT 1
        FROM [SkiCards] AS [s]
        WHERE [s].[ApplicationUserId] = @__userId_0)
    THEN CAST(1 AS BIT) ELSE CAST(0 AS BIT)
END
```

Метод `POST` проверяет состояние модели представления, анализируя значение `ModelState.IsValid`. Проверка состояния модели более подробно рассматривается в главе 11. Если модель действительна, создается новый экземпляр `SkiCard`, у которого `ApplicationUserId` присваивается текущее значение `userId`, а `CreatedOn` присваивается `DateTime`. Остальные свойства задаются на основании значений, переданных через экземпляр модели представления. Наконец, карта добавляется в `DbSet SkiCards`, а вызов `SaveChangesAsync` сохраняет изменения в базе данных.

После завершения сохранения пользователь перенаправляется к методу действия `Index`, где он видит, что новая карта появилась в списке.

```
// POST: SkiCard/Create
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Create(CreateSkiCardViewModel viewModel)
{
    if (ModelState.IsValid)
    {
        var userId = _userManager.GetUserId(User);

        SkiCard skiCard = new SkiCard
        {
            ApplicationUserId = userId,
            CreatedOn = DateTime.UtcNow,
            CardHolderFirstName = viewModel.CardHolderFirstName,
            CardHolderLastName = viewModel.CardHolderLastName,
            CardHolderBirthDate = viewModel.CardHolderBirthDate.Value.Date,
            CardHolderPhoneNumber = viewModel.CardHolderPhoneNumber
        };

        _skiCardContext.SkiCards.Add(skiCard);
        await _skiCardContext.SaveChangesAsync();

        return RedirectToAction(nameof(Index));
    }

    return View(viewModel);
}
```

Методы действия Edit

Методы действия `Edit` следуют той же схеме «POST/Redirect/GET», что и методы действия `Create`. Основное отличие заключается в том, что метод `Edit` получает один параметр `Id`, который определяет идентификатор редактируемой записи карты.

Необходимо принять особые меры для того, чтобы пользователи не могли редактировать данные карт других пользователей. Для этого при выборке редактируемых карт можно добавить фильтр по текущему экземпляру `ApplicationUserId`. Если ни один результат не найден, команда `return NotFound()` возвращает браузеру ошибку 404.

```
// GET: SkiCard/Edit/5
public async Task<ActionResult> Edit(int id)
{
    var userId = _userManager.GetUserId(User);
```

```

var skiCardViewModel = await _skiCardContext.SkiCards
    .Where(s => s.ApplicationUserId == userId && s.Id == id)
    .Select(s => new EditSkiCardViewModel
    {
        Id = s.Id,
        CardHolderFirstName = s.CardHolderFirstName,
        CardHolderLastName = s.CardHolderLastName,
        CardHolderBirthDate = s.CardHolderBirthDate,
        CardHolderPhoneNumber = s.CardHolderPhoneNumber
    }).SingleOrDefaultAsync();

if (skiCardViewModel == null)
{
    return NotFound();
}

return View(skiCardViewModel);
}

```

При создании `EditSkiCardViewModel` для `SkiCard` используется проекция. `EditSkiCardViewModel` не содержит все свойства сущности `SkiCard`, потому что свойства `ApplicationUserId` и `CreatedOn` используются только для чтения и не отображаются на форме редактирования.

Метод действия `POST` сначала проверяет состояние модели на действительность. Если модель представления действительна, сначала загружается заданная запись `SkiCard`. И снова фильтр возвращает только карты для текущего пользователя. Если карта не найдена, метод `NotFound()` возвращает ответ 404.

Если указанная карта найдена и принадлежит текущему пользователю, измените свойства `SkiCard` в зависимости от значений, переданных методу `Edit` через модель представления, и вызовите `SaveChangesAsync` в контексте карты. В этот момент система отслеживания изменений EF обнаруживает любые внесенные изменения и генерирует команду обновления для записи базы данных. После того как сохранение будет завершено, пользователь перенаправляется к методу действия `Index`.

```

// POST: SkiCard/Edit/5
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Edit(EditSkiCardViewModel viewModel)
{
    if (ModelState.IsValid)
    {
        var userId = _userManager.GetUserId(User);

        var skiCard = await _skiCardContext.SkiCards
            .SingleOrDefaultAsync(s => s.ApplicationUserId == userId &&
                s.Id == viewModel.Id);
    }
}

```

```
        if (skiCard == null)
        {
            return NotFound();
        }

        skiCard.CardHolderFirstName = viewModel.CardHolderFirstName;
        skiCard.CardHolderLastName = viewModel.CardHolderLastName;
        skiCard.CardHolderPhoneNumber = viewModel.CardHolderPhoneNumber;
        skiCard.CardHolderBirthDate = viewModel.CardHolderBirthDate.Value.Date;

        await _skiCardContext.SaveChangesAsync();

        return RedirectToAction(nameof(Index));
    }
    return View(viewModel);
}
```

Когда контекст базы данных и контроллер будут готовы, приложение начнет обрабатывать запросы, связанные с созданием и редактированием карт. Создание представлений, относящихся к контроллеру, рассматривается в главе 11.

Типы абонементов

Следующий ограниченный контекст содержит информацию о разных типах абонементов. Он состоит из трех сущностей: `PassType`, `PassTypeResort` и `PassTypePrice`. Сущность `PassType` состоит из имени и описания, а также других свойств, описывающих, когда и на каких курортах могут использоваться абонементы. Сущность `PassTypePrice` определяет цену конкретного вида `PassType` для клиентов разных возрастов. Например, стандартный дневной абонемент стоит дороже для взрослых и дешевле для детей.

```
public class PassType
{
    public PassType()
    {
        PassTypeResorts = new List<PassTypeResort>();
        Prices = new List<PassTypePrice>();
    }

    public int Id { get; set; }

    [MaxLength(255)]
    [Required]
    public string Name { get; set; }

    public string Description { get; set; }
```

```
public DateTime ValidFrom { get; set; }

public DateTime ValidTo { get; set; }

/// <summary>
/// Максимальное количество активаций абонемента данного типа.
/// Например, стандартный дневной абонемент может активироваться
/// только 1 раз. Годовой абонемент может активироваться до 265 раз
/// (количество дней, в которые курорт открыт в течение года).
/// </summary>
public int MaxActivations { get; set; }

public List<PassTypeResort> PassTypeResorts { get; set; }

public List<PassTypePrice> Prices { get; set; }
}

public class PassTypeResort
{
    public int ResortId { get; set; }

    public int PassTypeId { get; set; }
}

public class PassTypePrice
{
    public int Id { get; set; }

    [Range(0, 120)]

    public int MinAge { get; set; }

    [Range(0, 120)]
    public int MaxAge { get; set; }

    public decimal Price { get; set; }

    public int PassTypeId { get; set; }
}
```

Класс `PassTypeContext` содержит один набор `DbSet` для `PassType`. Определять `DbSet` для `PassTypeResort` или `PassTypePrice` не нужно, потому что все операции с этими двумя сущностями выполняются через экземпляры сущности `PassType`. При этом EF распознает сущности `PassTypeResort` и `PassTypePrice` как часть этого контекста из-за отношений, определенных для сущности `PassType`.

Первичным ключом для сущности `PassTypeResort` является составной ключ, заданный с использованием `ModelBuilder` в методе `OnModelCreating` контекста.

`ModelBuilder` может использоваться для определения дополнительной информации о первичных ключах, внешних ключах, типах свойств и других ограничениях. В общем случае соглашений, встроенных в EF, достаточно для ваших сущностей, но время от времени необходимо переопределять эти соглашения в методе `OnModelCreating`.

```
public class PassTypeContext : DbContext
{
    public PassTypeContext(DbContextOptions<PassTypeContext> options)
        :base(options)
    {
    }

    public DbSet<PassType> PassTypes { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<PassTypeResort>()
            .HasKey(p => new { p.PassTypeId, p.ResortId });

        base.OnModelCreating(modelBuilder);
    }
}
```

Административное приложение Alpine Ski House в будущем будет предоставлять механизм для создания типов абонементов. А для этого спринта команда Alpine Ski House решила воспользоваться миграцией для вставки основных типов абонементов.

Абонементы и проверка

Последним ограниченным контекстом, реализованным в этом спринте, является контекст абонементов. Абонементы связываются с картой: абонемент предоставляет держателю карты доступ к подъемникам определенного курорта в течение некоторого времени. Период времени и конкретный курорт зависят от свойств типа абонемента.

Первая версия ограниченного контекста для абонементов содержит три сущности: `Pass`, `PassActivation` и `Scan`. Сущность `Pass` представляет конкретный тип абонемента (`TypeId`), приобретенного для конкретной карты (`CardId`). Сущность `Scan` представляет каждое отдельное сканирование карты в разных местах курорта. Сущность `PassActivation` представляет использование абонемента на курорте в конкретный день. Абонемент активируется сервисом проверки при первом сканировании карты на подъемнике в конкретный день. Для каждого `SkiPass` разрешается определенное количество активаций абонементов, определяемое

свойством `MaxActivations` соответствующей сущности `PassType`. Если на карте клиента не осталось активаций, клиент не допускается к подъемнику.

Схема определения этих классов сущностей и их класса `DbContext` к этому моменту должна выглядеть знакомо.

```
public class Pass
{
    public Pass()
    {
        this.Activations = new List<PassActivation>();
    }

    public int Id { get; set; }

    public int CardId { get; set; }

    public int PassTypeId { get; set; }

    public DateTime CreatedOn { get; set; }

    public List<PassActivation> Activations { get; set; }
}

public class PassActivation
{
    public int Id { get; set; }

    public int PassId { get; set; }

    public int ScanId { get; set; }

    public Scan Scan { get; set; }
}

public class Scan
{
    public int Id { get; set; }

    public int CardId { get; set; }

    public int LocationId { get; set; }

    public DateTime DateTime { get; set; }
}

public class PassContext : DbContext
{
    public PassContext(DbContextOptions<PassContext> options)
        :base(options)
```



```
{  
}  
  
public DbSet<Pass> Passes { get; set; }  
  
public DbSet<PassActivation> PassActivations { get; set; }  
  
public DbSet<Scan> Scans { get; set; }  
}
```

События и обработчики событий

Клиенты могут покупать абонементы на сайте в интернете, но обработка платежей и связанный с ними ограниченный контекст выходят за рамки этого спринта. Моделируя взаимодействия между ограниченными контекстами как события, вы можете реализовать логику создания абонементов без создания полноценной системы обработки платежей.

При завершении покупки ограниченный контекст покупок генерирует событие. Это событие предоставляет информацию о типах приобретенных абонементов для разных типов карт. Ограниченный контекст абонементов должен создавать новые сущности *SkiPass* при каждом возникновении этого события.

Для решения этой задачи используется паттерн передачи сообщений «публикация/подписка»: одна сторона создает уведомления о событиях и публикует их на шине передачи сообщений. Затем шина передачи сообщений оповещает всех подписчиков, заинтересованных в этом конкретном типе оповещений о событиях. Преимущество этого паттерна заключается в том, что публикующей стороне не нужно ничего знать о подписчиках. Паттерн способствует формированию структуры приложения, он прост для понимания, тестирования и расширения. В данном случае он также позволяет реализовать логику в ограниченном контексте абонементов без полностью функционирующей системы покупки и оформления заказа.

В этом примере для реализации данного паттерна в проекте Parsley используется библиотека MediatR¹. MediatR обеспечивает простую внутрипроцессную передачу сообщений в приложениях .NET. События представляются *POCO*-классами, реализующими интерфейс *INotification*. Событие *PurchaseCompleted* генерируется ограниченным контекстом покупок после завершения процесса оформления заказа. В нем содержится вся необходимая информация, относящаяся к покупке.

```
public class PurchaseCompleted : INotification  
{  
    public string UserId { get; set; }  
    public DateTime Date { get; set; }  
    public string TransactionId { get; set; }  
}
```

¹ <https://github.com/jbogard/MediatR>.

```

        public decimal TotalCost { get; set; }
        public List<PassPurchased> Passes { get; set; }
    }
    public class PassPurchased
    {
        public int PassTypeId { get; set; }
        public int CardId { get; set; }
        public decimal PricePaid { get; set; }
        public string DiscountCode { get; set; }
    }

```

В реализации ограниченный контекст покупок публикует экземпляр класса `PurchaseCompleted` с использованием шины `MediatR`.

```
_bus.Publish(purchaseCompletedEvent);
```

В случае ограниченного контекста абонементов нас интересуют в основном свойства `PassTypeId` и `CardId` каждого экземпляра `PassPurchased`. Чтобы подписаться на событие `PurchaseCompleted`, создайте класс, реализующий интерфейс `INotificationHandler<PurchaseCompleted>`, и добавьте свою логику в метод `Handle`.

```

public class AddSkiPassOnPurchaseCompleted : INotificationHandler<PurchaseCompleted>
{
    private readonly PassContext _passContext;
    private readonly IMediator _bus;

    public AddSkiPassOnPurchaseCompleted(PassContext passContext, IMediator bus)
    {
        _passContext = passContext;
        _bus = bus;
    }

    public void Handle(PurchaseCompleted notification)
    {
        var newPasses = new List<Pass>();
        foreach (var passPurchase in notification.Passes)
        {
            Pass pass = new Pass
            {
                CardId = passPurchase.CardId,
                CreatedOn = DateTime.UtcNow,
                PassTypeId = passPurchase.PassTypeId
            };
            newPasses.Add(pass);
        }

        _passContext.Passes.AddRange(newPasses);
        _passContext.SaveChanges();

        foreach (var newPass in newPasses)

```

```
{
    var passAddedEvent = new PassAdded
    {
        PassId = newPass.Id,
        PassTypeId = newPass.PassTypeId,
        CardId = newPass.CardId,
        CreatedOn = newPass.CreatedOn
    };
    _bus.Publish(passAddedEvent);
}
}
```

Обработчик сначала создает новые экземпляры класса `Pass` на основании информации, переданной в уведомлении о событии. Затем новые абонементы добавляются в контекст абонементов, а изменения сохраняются в базе данных. Наконец, обработчик публикует событие `PassAdded` для каждого созданного абонемента. В настоящее время для события `PassAdded` обработчиков нет, но такая реализация создает простую точку расширения в приложении. Любая логика, инициируемая при добавлении абонемента, описывается в виде класса, реализующего `INotificationHandler<PassAdded>`.

Экземпляр `PassContext` и `IMediator` передается конструктору обработчика. Эту задачу решает встроенный механизм внедрения зависимостей исполнительной среды ASP.NET Core, рассматриваемый в главе 14.

Итоги

В Entity Framework Core компания Microsoft предоставила простой API для загрузки и сохранения информации в базе данных. Многих распространенных проблем, присущих ORM, можно избежать разбиением больших моделей данных на меньшие, более управляемые ограниченные контексты. Хотя EF Core — не единственная библиотека для работы с данными, доступная для ASP.NET Core, во многих приложениях ее следует рассматривать как вариант по умолчанию.

В следующей главе мы рассмотрим механизм представлений Razor.

11

Представления Razor

Кэндис стучала по клавишам и что-то напевала под музыку в плеере. Она была поглощена работой и трудилась над рефакторингом представлений, чтобы использовать то новое, что она узнала о Razor. Она не слышала, как Честер подошел сзади и спросил, почему она до сих пор на работе. И конечно, когда он заглянул ей через плечо, она вовсе не собиралась заехать ему прямо в челюсть, отчего Честер повалился на ковер.

«О господи! — воскликнула она. — Ради всего святого, что ты творишь?!»

«Вообще-то мне больно, Кэндис, — ответил Честер с пола, разглядывая подвесной потолок. — За что это?»

«Ты напугал меня, дубина. А у меня черный пояс по тхэквондо, знаешь ли. Не стоит ко мне так подкрадываться». Встав из-за компьютера, чтобы помочь Честеру подняться, она поняла, что за окнами стемнело, а в офисе больше никого нет. — «А сколько сейчас времени?»

«Ууу, — простонал он. — Больно, Кэндис».

«Да, я знаю. Извини. Да вставай уже. — Она помогла Честеру подняться и села обратно. — Конец рабочего дня? Запираешь?»

«Нет, Тим еще здесь. Я ему на тебя пожалуюсь, — усмехнулся он. — Если останется синяк, с тебя обед. А что ты здесь делаешь? Обычно ты уходишь еще час назад».

«Я знаю, просто засиделась с представлениями. С картами закончила чуть раньше, чем ожидалось. Мы с Даниэль на этой неделе разбирались с Razor — клевая штука. Очень, очень умно. Я поговорила с Балашем, и он выделил мне пару дней на то, чтобы немного почистить представления».

«Ага, так наш программист Ruby меняет убеждения? Пересаживается в лодку Microsoft?»

«До такого еще далеко, — рассмеялась Кэндис. — Но да, им надо отдать должное. Экономит много времени. Помнишь, сколько проблем было со всей этой размет-

кой? А теперь посмотри сюда. — Она открыла код представления для создания карт. — Ты уже видел тег-хелперы?»

«Нет. В смысле — да, проблемы я помню, но тег-хелперы еще не видел».

«Ладно, хватай кресло, и я тебе покажу, что такое Razor... но тогда, чур, обед с тебя. Мы посидим и вспомним, как ты пытался ко мне подкрасться».

Создание веб-сайтов с точки зрения современного разработчика

Один из самых интересных аспектов современной веб-разработки заключается в том, что мы уже не пишем разметку HTML напрямую, хотя HTML остается важнейшей частью работы пользователя в браузере. Звучит невероятно, но это так. Конечно, фрагменты HTML встречаются там и сям, но что же мы делаем на практике? Мы пишем приложения, генерирующие разметку HTML, так что в действительности HTML является скорее побочным продуктом работы, но не прямым ее результатом.

В наши дни применяются различные инструменты, предназначенные для использования контекста запроса пользователя и преобразования его в форму, которая может интерпретироваться браузером. Но поскольку HTML не является языком программирования, мы безусловно не можем сказать, что наши приложения пишутся на HTML. Даже в одностраничных приложениях разработчики интерфейсной части с большей вероятностью пишут шаблоны и занимаются привязкой в коде, чем написанием чистой разметки HTML. Мы используем инструменты, шаблоны и языки, которые больше подходят для взаимодействия с кодом на сервере, и именно в их выводе (с точки зрения браузера) встречается разметка HTML.

Существует много способов генерации вывода HTML из C#, потому что HTML в конечном итоге представляет собой строку. Строки можно объединять для создания HTML:

```
var html = "<html>";
html += "<body>";
html += "<div class=\"exciting\">Hello" + name + "!</div>";
html += "</body>";
html += "</html>";
```

Однако этот метод выглядит уродливо и создает невероятные трудности с сопровождением. Пожалуй, с новым механизмом интерполяции строк C# 6 код станет немного симпатичнее.

```
var html = @$"<html>
    <body>
```

```
        <div class='exciting'>Hello {name}!</div>
    </body>
</html>";
```

Выглядит получше, чем предыдущая версия, и все же было бы еще лучше иметь способ построения HTML, который поддерживает IntelliSense и предотвращает типичные ошибки (например, когда разработчик забывает об HTML-кодировании строк). И такой инструмент существует: Razor.

Использование предыдущих достижений и находок

Стоит отметить, что за механизмом представлений Razor лежит богатая история разработки, базирующаяся на информации, однако Razor отходит от подхода предыдущих разработок по принципу «один файл на страницу». Razor отходит от обязательной, явной системы записи с угловыми скобками и знаками %, используемой в веб-формах, и упрощает повторное использование ранее выполненной работы (если Razor используется так, как было задумано разработчиками). Разработчики сознательно решили построить синтаксис разметки шаблонов, который позволяет сплести код C# с HTML и который будет знаком каждому, кто уверенно работает на языках .NET. Razor также обеспечивает поддержку IntelliSense, кодирования HTML и всех остальных синтаксических удобств, а также предоставляет базовый набор вспомогательных средств, которые делают рабочий процесс разработки представлений более динамичным по сравнению с предшественниками.

Razor продолжает расти и совершенствоваться с каждой версией. Технология была представлена в эпоху MVC 3 Framework, а первая версия была выпущена в июне 2010 года.

Роль Razor

Razor — используемый по умолчанию и единственный зрелый механизм представлений ASP.NET Core MVC, однако смысл определения Razor несколько размылся из-за разнообразия удобств, предоставляемых проектам. Когда речь заходит о Razor, с большой вероятностью имеется в виду синтаксис, содержащийся в представлении (а может быть, сам файл). Но Razor также включает парсер и систему создания маркеров (tokenizer) для обработки синтаксиса, средства редактирования для поддержки IntelliSense в Visual Studio и генератор C# с множеством разнообразных генераторов блоков (chunks) для построения файла класса C#, соответствующего вашему представлению.

Одни из этих возможностей работают во время редактирования, другие активизируются во время выполнения, и хотя все компоненты играют важную роль в том, чтобы редактирование было удобным, а приложение работало как положено, нас как разработчиков сейчас интересует синтаксическая сторона Razor и некоторые из используемых соглашений.

Razor позволяет при создании представления переключаться с HTML на C# и обратно гладко и почти как по волшебству. Razor неявно учитывает контекст элементов HTML и правила типизованного языка C#, но при этом также позволяет устанавливать явные границы для обозначения того, какие части содержат разметку, а какие — код. Вы можете создавать шаблоны на основе своих страниц, включать необязательные части по определенным вами условиям или подниматься на уровень самого представления.

Конечная цель Razor — предоставить вам механизм управления визуализацией представления; не забывайте об этом важном факте. Представления не предназначены для хранения бизнес-правил и логики проверки. Вы не должны инициировать в них подключения, обрабатывать события или заходить на несколько уровней условной логики для принятия решения о том, должна ли включаться в визуализацию некоторая часть вашего представления или обращаться с вызовом к сервисам. Такой код становится хрупким, трудным в сопровождении, а часть бизнес-логики изолируется в коде, который трудно тестировать. Совет от ключевого участника проекта Razor:

Razor — всего лишь язык. Старайтесь не включать слишком много логики в само представление. Абстрагируйте ее в компоненты и модули, если это возможно.

Тейлор Маллен (Taylor Mullen), один из разработчиков группы Razor в Microsoft.

Мораль: не усложняйте. Хотя технология Razor позволяет делать некоторые сложные, впечатляющие вещи, на самом деле она создавалась для того, чтобы помочь вам в создании представлений — простых, удобочитаемых и правильно функционирующих при получении правильных входных данных.

Основы Razor

Как вы помните из главы 3 «Модели, представления и контроллеры», представление отвечает за часть опыта взаимодействия, обращенного к пользователю. Представление часто содержит разметку HTML, которая в свою очередь может содержать ссылки на CSS и JavaScript, но представление невозможно напрямую просмотреть в браузере. Вспомните, что ваше представление превращается в класс C#. Даже если вы не пишете код C# напрямую, то, что вы пишете, будет преобразовано в код C#.

Взгляд «за кулисы»

Внутреннее устройство механизма представлений Razor в целом остается невидимым для вашего проекта. Собственно, многим разработчикам его глубокое знание и не требуется, но если вы взглянете на то, что происходит «за кулисами», происходящее на сцене перестанет казаться волшебным. В общем, если вы предпочита-

ете оставить фокусника в покое и не заставлять его раскрывать секреты — этот раздел не для вас!

Вы уже видели в этой книге (и вероятно, в прилагаемом проекте) ряд представлений и содержащийся в них код. Если вы немного поэкспериментировали и «сломали» представление (а эксперименты такого рода в высшей степени полезны!), возможно, вам также попадалось сообщение об ошибке с указанием номера строки. Если остановиться и задуматься, это выглядит немного странно, не так ли? Вы видите нечто похожее на рис. 11.1 — кажется, ваше представление содержит ошибку компиляции.

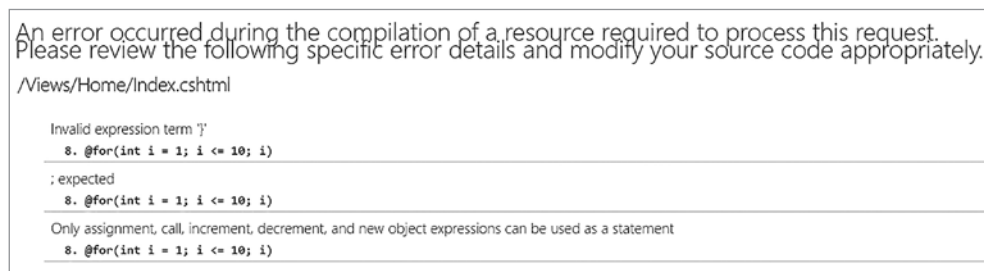


Рис. 11.1. Пример сообщения об ошибке при компиляции представления Razor

В сообщении указывается ряд возможных причин ошибки; виновником здесь является отсутствие оператора инкремента для переменной `i` в команде `for`. Компилятор указывает правильный номер строки — источника ошибки — для нашего представления, а не из сгенерированного класса представления C#, потому что он дает гораздо меньше полезной информации для разработчиков.

А вот как это работает: в ходе разбора ваше представление обрабатывается и разбивается на части. Некоторые из этих частей уже написаны на C#, другие разбиваются на фрагменты текста и вставляются в структуру, которая станет классом C#. Для примера возьмем код из листинга 11.1: этот простой блок из файла представления Razor содержит смесь C# и HTML и выводит сообщение.

Листинг 11.1. Пример синтаксиса Razor с кодом C# из гипотетического представления `SampleView.cshtml`

```
@for(int i = 1; i <= 5; i++) {  
    <p>This is index #@i</p>  
    i += 1;  
}
```

Чтобы мы могли получить осмысленную информацию от отладчика во время выполнения, «за кулисами» происходит много всего. В листинге 11.2 показано, во что превращается часть кода представления Razor из листинга 11.1 после разбора. Конечно, этот листинг не полон. Мы не стали включать команды `using`, простран-

ства имен, объявления классов или имена функций, потому что они занимают слишком много места, но перед вами основная суть кода Razor из листинга 11.1.

Листинг 11.2. Часть сгенерированного класса C# из представления Razor

```
#line 1 "SampleView.cshtml"
for(int i = 1; i <= 5; i++ ) {

#line default
#line hidden

        Instrumentation.BeginContext(19, 4, true);
        WriteLiteral("    <p>This is index #");
        Instrumentation.EndContext();
        Instrumentation.BeginContext(29, 1, false);
#line 2 "SampleView.cshtml"
            Write(i);

#line default
#line hidden
            Instrumentation.EndContext();
            Instrumentation.BeginContext(31, 6, true);
            WriteLiteral("</p>\r\n");
            Instrumentation.EndContext();
#line 3 "SampleView.cshtml"
}
}
```

Вы видите как сам код C#, так и вызовы `WriteLiteral` для добавления HTML в документ. Также здесь присутствуют вызовы начала и завершения инструментальных контекстов, но, пожалуй, самое интересное — это директивы `#line`, отмечающие файл-источник и номер строки, которой соответствует код. Именно эти директивы позволяют выдавать ошибки компиляции во время выполнения с информацией, соответствующей представлению.

Да, все верно: это происходит во время выполнения, и это еще одно отступление от предыдущих версий MVC Framework. Представления Razor компилируются в классы C# в памяти в виде строки. Строка передается сервису компиляции MVC, который компилирует представление в другой артефакт, хранящийся в памяти, — сборку представления, а исполнительная среда затем загружает результат в память.

ПРИМЕЧАНИЕ

На момент написания книги поддержка компиляции представлений View в ASP.NET Core MVC на стадии сборки была возможна лишь с применением обходных решений и изрядного труда. А это означает, что представления нельзя преобразовать в классы и разместить в отдельных сборках, которые могут загружаться в проект извне. Тем не менее специалисты из группы ASP.NET говорят, что они работают над решением для тех, кому нужна такая функциональность.

Представление как часть сборки может использоваться как любой другой класс. При неявном запросе (при использовании `View()` для контроллера) или явном запросе (по имени) сервис обнаружения представлений обрабатывает тип, лежащий в его основе, асинхронно вызывает сгенерированный метод.

Вы лучше поймете, что происходит «за кулисами», когда мы завершим свое первое представление и попробуем запустить его. Но сначала нужно разобраться с тем, что необходимо знать для создания представлений, и в частности конструкций, которые могут использоваться для управления механизмом представлений и обеспечения полного доступа к представлению как к классу `.NET`, которым оно является.

Выражения в синтаксисе Razor

По умолчанию Razor считает, что вы работаете с HTML; прекрасно, если вы занимаетесь только написанием разметки. Даже если большинство пользователей не понимают, что где-то незаметно для них работает приложение, они все равно рассчитывают увидеть нечто актуальное для себя. Если вы хотите чередовать статические части с более содержательной контекстной информацией, которая сделает страницу актуальной для пользователя, нужно каким-то образом сообщить Razor, что далее идет код `C#`. Возьмем следующую конструкцию:

```
<p>Today is @DateTime.Today.ToString("D"). What a wonderful day.</p>
```

Для этой команды генерируется следующая разметка HTML:

```
<p>Today is October 13, 2016. What is a wonderful day.</p>
```

Сначала идет обычная разметка HTML, а после нее внедряется отформатированная дата с использованием символа `@`, который сообщает Razor о начале программного кода. Здесь интересно то, что за вызовом `ToString` следует точка, однако парсер в этом случае может определить, что программа не пытается вызывать метод или обратиться к свойству. Код такого типа называется неявным выражением.

Если неявный вызов не подходит (например, если вы пытаетесь выполнить вычисления или начать код с неочевидного элемента `C#`), можно воспользоваться синтаксисом явного выражения:

```
<p>Half of 10 is @(10 / 2).</p>
```

Результат выглядит так, как и следовало ожидать:

```
<p>Half of 10 is 5.</p>
```

Чтобы создать явное выражение, заключите свой код в круглые скобки; закрывающая круглая скобка сигнализирует о том, что вы намерены вернуться обратно к HTML. Как в явном, так и в неявном механизме может использоваться только

одно выражение. Если вы хотите включить в разметку большой объем кода, вероятно, для этого лучше воспользоваться блоком кода.

Переключение в режим кода

Ранее в листинге 11.1 был приведен код с командой `for`, в котором происходило переключение на разметку HTML. Весь этот код как единое целое был заключен в символы `@{..}`, а полученная конструкция называется блоком кода. Давайте ненадолго задержимся на листинге 11.3 с большим объемом кода.

Листинг 11.3. Расширенный блок кода с дополнительными командами C#

```
@for (int i = 1; i <= 5; i++)
{
    <p>This is index @(i)</p>
    for (int j = 0; j < 10; j++)
    {
        if (j >= 5)
        {
            <p>Multiplied by the inner index, the product is @(j * i).</p>
        }
    }
    i += 1;
}
```

Самое замечательное в этом фрагменте то, что наш код чрезвычайно гибок, и мы по-прежнему можем легко переключаться между разметкой HTML и командами C#. Однако блок кода отличается тем, что парсер ожидает видеть в нем код C# вместо разметки HTML, потому что вне блока также находится C#. Парсер действует здесь более строго, потому что у него нет выбора. Вход в тег HTML для парсера очевиден, потому что конструкция `<p>` не является кодом C#, но для выхода из блока HTML тег должен быть завершен явно. HTML не требует наличия завершающего тега, и за пределами блока кода в файле Razor это правило остается истинным, но если тег открывается внутри блока кода, не забудьте помочь Razor и закрыть его.

И снова возврат в HTML происходит довольно просто: достаточно обернуть контент в тег, как в случае с тегом `<p>` в ранее описанном коде. Поскольку механизм представлений Razor ожидает, что по умолчанию в блоке кода используется C#, важно заметить, что в этом случае включение обычного текста без разметки (как в листинге 11.4) становится недопустимым.

Листинг 11.4. Недопустимый символьный контент в блоке кода

```
@if(true)
{
    Hello.
}
```

Этот код недопустим, потому что `Hello.` внутри блока кода не является действительной командой C#. Но если бы `Hello` или другой набор символов также встречался в классе приложения, то парсер не смог бы определить, идет ли речь о строковом литерале `Hello` или о попытке обращения к свойству класса `Hello`. На первый взгляд кажется, что это создает серьезные трудности при попытке добавления контента в блоке кода, но и эта проблема может быть решена явным переключением в режим разметки.

Явное использование разметки

Код из листинга 11.4 можно исправить, упаковав приветствие в тег `<text>`. В листинге 11.5 продемонстрированы два способа, которые позволяют парсеру интерпретировать сообщение именно так, как мы запланировали.

Листинг 11.5. Примеры правильного обозначения текста в блоке кода

```
@if(true)
{
    <text>Hello.</text>
}

@if(true)
{
    @:Hello.
}
```

Тег `<text>` является частью синтаксиса Razor, это не какая-то вновь изобретенная разметка HTML. Теги `<text>` обрабатываются на стороне сервера и исключаются из ответа, возвращаемого клиенту. Весь текст, помещенный внутри тега, воспроизводится так, как если бы он находился вне блока кода. Теги `<text>` могут занимать несколько строк.

В каком-то смысле последовательность символов `@:` может рассматриваться как сокращение для тегов `<text>`. Единственное различие заключается в том, что он только приказывает парсеру интерпретировать остаток текста в строке как обычный текст. Если текст переходит на следующую строку, то парсер снова считает, что все, что вы написали во второй строке, является кодом C#.

Памятка по управлению парсером Razor

Кроме основных возможностей, упоминавшихся ранее, есть и другие скрытые синтаксические удобства. В табл. 11.1 приведены краткие описания таких возможностей, объединенные в удобную памятку.

Таблица 11.1. Памятка по управлению парсером Razor

Синтаксис	Описание
@	Начало неявного выражения с переключением в режим кода C#; продолжается до первого символа, не являющегося символом C#
@@	Escape-последовательность для символа @; не обязательна при работе с адресами электронной почты, так как Razor распознаёт структуру адреса и автоматически интерпретирует адрес как текст
@()	Явное выражение; код в круглых скобках интерпретируется как код C#
@{ ... }	Явный блок кода; всё в фигурных скобках интерпретируется как код C#, если только другая управляющая последовательность не переведет парсер в режим текста
<text>...</text>	Серверный тег, который не передается на сторону клиента; все содержимое тега включается в ответ как текст
@:	Явное включение текстового режима до конца текущей строки
// /*...*/	Комментарии в стиле C#; действительны только при работе парсера в режиме C#, в текстовом режиме интерпретируются как литеральный вывод
@* ... *@	Комментарии в формате Razor: все содержимое игнорируется парсером и не включается в вывод клиентского ответа

Относительно всех этих команд управления необходимо запомнить один важный момент: вы всего лишь указываете Razor предпочтительный режим. Ничто не мешает вам переключаться между режимами во вложенных последовательностях. Однако здесь действует та же рекомендация, что и при написании кода: с ростом сложности представления растут и усилия, необходимые для его сопровождения. Если окажется, что в представлении что-то невозможно сделать легко, рассмотрите возможность перемещения части логики в модель представления, компонент представления или даже код JavaScript. Во всех этих случаях вы сможете ее нормально протестировать.

ПРИМЕЧАНИЕ

Группе ASP.NET Monsters удалось побеседовать с Тейлором Малленом (Taylor Mullen) — разработчиком, написавшим значительную часть кода механизма представления Razor в ASP.NET Core MVC. Посмотрите на канале Microsoft Channel 9 эпизод, в котором группа Monsters беседует с Тейлором о механизме представлений Razor: <https://channel9.msdn.com/Series/aspnetmonsters/ASPNET-Monsters-59-Razor-with-Taylor-Mullen>.

Использование других возможностей C#

Razor отнюдь не ограничивается простейшими синтаксическими функциями. Некоторые возможности также будут рассмотрены в книге позднее, особенно в главе 19 «Многоразовый код» и главе 21 «Расширение фреймворка».

Использование типов C# в представлениях

Как и в любом другом блоке кода C#, необходимо ввести в контекст пространства имен и типы, с которыми вы собираетесь работать, при помощи команды `using`. Это могут быть как типы, определенные в вашей сборке, так и типы, вводимые через зависимости внешних библиотек.

```
@using AlpineSkiHouse.Models
```

Чтобы сохранить нормальную структуру кода, разместите команды `using` в начале файла. Впрочем, это не обязательно, потому что все команды `using` все равно перемещаются в начало сгенерированного класса. Вы можете отойти от рекомендаций C# и раскидать команды `using` по всему коду, но по соображениям удобства чтения и сопровождения лучше так не делать.

Определение модели

В Razor используется концепция «представлений с сильной типизацией»; это означает, что ваше представление знает, какому типу объекта оно соответствует. Это позволяет легко вплетать свойства модели в представление. Сначала необходимо сообщить Razor, какой тип вы собираетесь использовать:

```
@model SkiCard
```

После этого в странице можно обращаться к свойствам модели:

```
Welcome to your Ski Card summary, @Model.CardHolderFirstName.
```

Здесь может пригодиться ваше понимание средств управления парсером Razor. Наблюдательный читатель заметит неявное выражение в середине блока текста и поймет, что механизм представлений обрабатывает его соответствующим образом.

Хотя тип используемой модели может определяться в самом представлении, в конечном итоге контроллер создает экземпляр модели (или обеспечивает его создание) и передает его представлению. Кроме того, контроллер должен убедиться в том, что тип, переданный представлению, совпадает с ожидаемым; в противном случае во время выполнения произойдет исключение.

Использование данных представления

Если подходить к процессу разработки со здоровым прагматизмом, приходится признать, что академически «правильный» подход не всегда годится для конкретной ситуации. Легко заявить, что «везде должна использоваться сильная типизация» или «для передачи данных от контроллера к представлению должны использоваться модели представления», но на практике иногда действуют другие факторы. Для примера возьмем текст, отображаемый в заголовке веб-страницы. Как правильно задать его? Разумеется, не в шаблоне, потому что текст заголовка может изменяться от страницы к странице. Назначение текста в представлении выглядит разумно, но что если в вашем приложении используется динамическое представление, которое должно отображать разные заголовки для разных типов просматриваемых элементов? Получается, нужно иметь модель представления, свойства которой могут задаваться контроллером, и передавать ее представлению. Тогда можно потребовать, чтобы модель представления обязательно включалась в каждую страницу. А если модель представления — что-то вроде типа `SkiCardViewModel`? Текст заголовка не имеет никакого отношения к модели представления `SkiCardViewModel`, так почему он должен в ней размещаться? И где лучше всего хранить текст заголовка в других сценариях — например, при использовании такой модели, как `IEnumerable<SkiCardViewModel>`?

Короче говоря, единственно правильного места может и не быть (по крайней мере не во всех ситуациях). Возможно, вы сможете побеседовать с другим разработчиком и найти хорошее решение, которое подходит вам обоим, но тут в комнату входит еще один разработчик, и вся работа идет насмарку. Сколько людей, столько и мнений, и это нормально. Красота языка и фреймворка проявляется в том, что нам разрешается иметь разные точки зрения. По тому же принципу «не предполагать лишнего» ASP.NET Core MVC поддерживает идею «данных представления», к которым можно обращаться через пару свойств без использования объекта с сильной типизацией. Таким образом, вам предоставляется альтернативный механизм хранения данных в контроллере и их загрузки в представлении без излишних жестких привязок. Использование механизма данных представления Razor редко приводит к проблемам, особенно в тех случаях, когда нужно передать только одно значение.

В простейшей форме в действии контроллера выполняется присваивание следующего вида:

```
ViewData["Message"] = "Thanks for reaching out.";
```

После этого данные читаются в представлении:

```
<h3>@ViewData["Message"]</h3>
```

Данные представления хранятся в объекте `IDictionary<string, object>`, поэтому сохранить можно все, что угодно. Механизм работает как для простых типов (таких, как `int` и `string`), так и для более сложных объектов — например, `SkiCardViewModel`. На словарь накладывается другой механизм для обращения к данным представления через динамическое свойство представления с именем `ViewBag`. При помощи `ViewBag` можно обращаться ко всем ключам словаря так, как если бы они были свойствами:

```
<h3>@ViewBag.Message</h3>
```

В обоих случаях при отсутствии в словаре элемента с заданным ключом вы не получите сообщения об ошибке при последующих попытках вывода данных представления.

Попытавшись обратиться к значению (скажем, включить объект в текст страницы), вы увидите, что Razor поступает так же, как поступил бы метод `Console.WriteLine`: вызывает метод `ToString()` объекта. Чтобы обратиться к свойствам объекта, лучше создать в начале представления переменную, область видимости которой совпадает с представлением:

```
@{  
    var skiCard = (SkiCardViewModel) ViewData["SkiCard"];  
}
```

Помните, что в конечном итоге Razor сгенерирует код C#, поэтому при неосторожности вы по-прежнему столкнетесь с такими проблемами, как исключения при использовании `null`-ссылок и преобразования типов. И хотя данные представлений могут предоставить простой механизм обмена данными, необходимо привести несколько рекомендаций, относящихся к использованию данных представлений.

Во-первых, предыдущий пример с сохранением сложного объекта в данных представления должен был показать, как преобразование типов работает в представлении — точно так же, как и везде. Возможно, вы найдете ситуацию, в которой будет разумно использовать этот вариант, но для передачи полной модели представления из контроллера в представление, очевидно, лучше использовать синтаксис `@model`. Если страница состоит из многих частей с разными моделями представлений, поддерживающими каждую часть, лучше присмотреться к компонентам представлений (глава 19).

Во-вторых, для принятия решения о том, стоит ли использовать данные представления, подумайте — относятся ли сохраняемые данные к модели или же находятся в компетенции представления? Как уже говорилось ранее, хранить модель представления в данных представления нежелательно; по тем же причинам не стоит разбивать объект на свойства и сохранять их по отдельности. Другой пример того, чего следует избегать:


```
ViewData["SlopeName"] = "Knee Bender";  
ViewData["Altitude"] = "3317m";  
ViewData["SnowDepth"] = "12dm";
```

Конечно, нам не нравится приводить код, который показывает, чего делать не стоит. Однако мы считаем, что важно знать, как выглядят неудачные решения, чтобы вы могли опознать потенциальные проблемы в своем коде. В блоке кода из нашего примера было бы странно полагать, что глубина снега (`SnowDepth`) на определенном склоне важна для представления для принятия решений или назначения текста заголовка в теге `<head>`. Следовательно, будет разумно сформировать модель представления, которая предоставляет эти свойства и передает их представлению через `@model`.

И наконец, не увлекайтесь использованием данных представления. Этот механизм был создан для соблюдения баланса между функциональностью, простотой сопровождения и удобством передачи простых данных веб-странице и обратно.

Работа с макетами

С первых дней веб-программирования нам приходилось решать проблему повторяющихся элементов на страницах: заголовков и футеров (подвал сайта), а также фирменной символики, которая должна была присутствовать на всех страницах сайта. Эти элементы достаточно легко программируются в первый раз, но многократное копирование со страницы на страницу затрудняет их синхронизацию. Ранние фреймворки позволяли включать такие компоненты на страницы в виде статических элементов.

По мере развития веб-технологий потребности веб-приложений также развивались. Включение файлов для таких аспектов, как заголовки и футеры, уже не справлялось с более сложными дизайнами — наподобие того, что изображен на рис. 11.2. Покупательские корзины, списки рекомендованных продуктов и управление учетными записями пользователей — все эти компоненты требовали генерации динамического содержания в зависимости от состояния текущего взаимодействия пользователя. Так пришлось развивать технологии шаблонов.

В ASP.NET 2.0 технология WebForms предоставила поддержку шаблонов в форме эталонных страниц (master pages) — чтобы построить страницу, разработчик брал внешний шаблон и заполнял его вложенными страницами. Такие страницы содержали специальные индикаторы и специализированные серверные теги, которые обеспечивали упаковку контента и гарантировали, что он будет вставлен в правильном месте.

В Razor используется несколько иной подход, потому что визуализация представления происходит раньше, чем визуализация макета. Это означает, что у вас появляется возможность присваивания значений (например, текста заголовка страни-

цы) через свойства данных представления с последующим использованием этих значений в макете. А теперь посмотрим, какой код необходим для формирования дизайна, показанного на рис. 11.2.

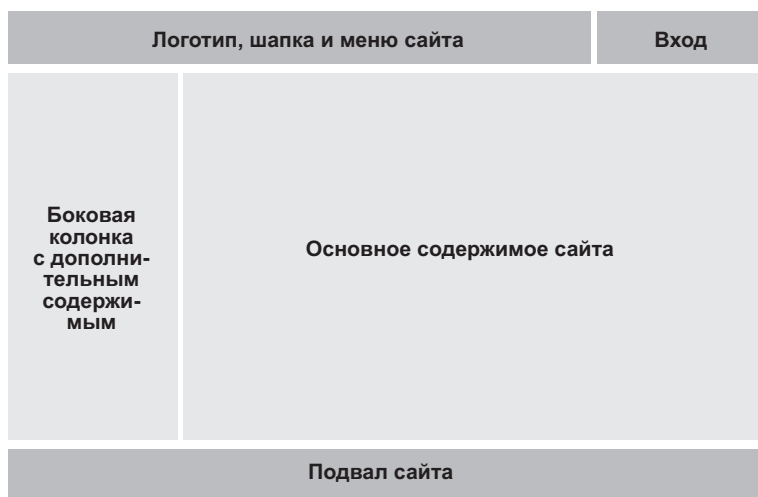


Рис. 11.2. Шаблон с интерактивными заменяемыми компонентами

Основы работы с макетами

Давайте посмотрим, как организовано взаимодействие макетов с представлениями. Вспомните, о чем говорилось в главе 3: контроллер может вернуть представление по соглашению, используя имя метода действия, или же явно запросить конкретное представление по имени. Само представление не обязано содержать большое количество макетной информации и не обязано задавать макет (хотя вы можете сделать и то и другое). Но если что-то из этого отсутствует, нужно определить, где его взять.

Построение механизма для шаблонов — нетривиальная задача, поэтому для управления значениями по умолчанию и местонахождением компонентов был создан набор соглашений, обеспечивающих работу макетов в Razor. Прежде всего нужно определить начальную точку для создания шаблонов. В папке **Views** присутствует файл `_ViewStart.cshtml`, который позволяет задать шаблон для использования в коде. В шаблоне проекта по умолчанию выбор макета происходит элементарно — примерно так:

```
@{  
    Layout = "_Layout";  
}
```

Это простое присваивание сообщает Razor, что в качестве шаблона используется файл с именем `_Layout.cshtml`. Согласно следующему соглашению использование `_ViewStart.cshtml` связано с местонахождением представлений. Любая папка, содержащая `_ViewStart.cshtml`, использует его в качестве отправной точки. Любая папка, не содержащая файла `_ViewStart.cshtml`, использует файл из ближайшего предка в дереве, с дальнейшим продвижением к корню папки `Views` в случае необходимости.

Как вы уже несомненно уяснили, Razor — это просто код C#, и `_ViewStart.cshtml` не является исключением. В этом файле можно писать код, оперировать данными представления или определять, какой макет должен использоваться по умолчанию в зависимости от конфигурации или бизнес-логики. Заданный здесь макет используется для визуализации любых представлений, но у вас имеется возможность переопределить его либо в контроллере, либо в самом представлении.

В листинге 11.6 приведено содержимое минимального макета для объединения основных компонентов на рис. 11.2. Двигаясь снаружи внутрь, мы видим определение DOCTYPE и внешний тег `<html>`, содержащий теги `<head>` и `<body>`.

Листинг 11.6. Пример файла макета, использующего ряд возможностей Razor

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewData["Title"] – My Site</title>
    <link href="/css/site.css" rel="stylesheet" />
</head>
<body>
    @await Html.PartialAsync("_Toolbar")
    <div class="container body-content">
        @RenderSection("sidebar", required: false)
        @RenderBody()
    </div>
    @Html.Partial("_Footer")
    @RenderSection("scripts", required: false)
</body>
</html>
```

Элемент `<title>` включает единственную конструкцию Razor в секции `<head>` — неявное выражение, которое вам уже встречалось ранее. Ничто не мешает включить в `<head>` больше информации. Собственно, мы еще вернемся к этому примеру позже в этой главе, когда будем рассматривать возможности его усовершенствования в контексте тег-хелперов.

Из прочих возможностей, используемых в этом примере, нас интересует прежде всего вызов `RenderBody`, но не беспокойтесь: мы также поговорим и о `PartialAsync`

с `RenderSection`. `RenderBody` отмечает место в документе, в котором Razor генерирует запрашиваемое представление.

Без вызова `RenderBody` произойдет ошибка времени выполнения — если только в шаблон не включается явный вызов `IgnoreBody`, с которым представление, запрашиваемое контроллером, генерироваться не будет.

ПРИМЕЧАНИЕ

Следует заметить, что даже при наличии вызова `IgnoreBody` ваше представление все равно будет выполнено. И хотя результат представления не возвращается клиенту в составе ответа, все ресурсы будут созданы, все сервисы будут вызваны, все компоненты представления будут выполнены — словом, отработают все операции, создающие побочные эффекты. Если вы решите, что в вашем решении следует использовать `IgnoreBody`, помните о любых нежелательных побочных эффектах: использовании серверных ресурсов, записи в журнал и подключениям к базам данных.

Включение секций из представлений

Если представление может рассматриваться как класс, то секция может рассматриваться как функция, вызываемая из родителя. Родитель определяет, где находятся секции, а потомки выбирают, что в них должно находиться. В примере из листинга 11.6 родителем будет макет, а представления сами решают, как реализовать секции. Любая определенная секция по умолчанию является обязательной, а Razor следит за тем, чтобы она присутствовала в каждом представлении на базе макета, в котором эта секция определена.

Включить секцию в представление относительно просто. После определения секции в макете, как это сделано в листинге 11.6, в представление добавляется код следующего вида:

```
@section sidebar {  
    <p>There are currently @cart.Count items in your cart.</p>  
    <nav>  
        <a asp-action="index" asp-controller="cart">View My Cart</a>  
        <a asp-action="index" asp-controller="checkout">Check Out and Pay</a>  
    </nav>  
}
```

Где бы эта секция ни находилась в представлении, она будет подставлена в то место, где она определяется в макете. Секция ничем не отличается от других частей страницы; в ней вы можете легко переключаться из текстового режима Razor в режим кода и обратно.

Все секции, определявшиеся до настоящего момента, существуют в макете. Секции можно определять на разных уровнях, но они не распространяются «вверх» по цепочке. Скажем, родитель в иерархии представлений ничего не знает о секции

ях, определяемых потомком. Вы можете определить секцию в макете и реализовать ее в представлении, но если, скажем, секция определяется в частичном представлении, обратиться к ней в макете вы не сможете.

Секции, встречающиеся в представлениях, должны определяться в макете, в котором происходит визуализация представления, если только секция не будет явно игнорироваться в макете вызовом `IgnoreSection("sectionName")`. Код, находящийся в игнорируемых секциях, не выполняется. Это может быть удобно, если вы хотите построить макет, в котором на каждой странице присутствует обязательная секция, которая может исключаться по некоторому определяемому вами критерию — например, в зависимости от того, ввел ли пользователь свои регистрационные данные.

Определение и потребление частичных представлений

Частичные представления предоставляют удобную возможность использования небольшой части шаблона или взаимодействия с пользователем между разными представлениями вашего приложения. Частичные представления можно считать чем-то вроде потомков представления, из которого выполняется их визуализация. Они отчасти напоминают пользовательские элементы управления WebForms и чем-то похожи на подключаемые файлы, которые использовались на заре веб-приложений. Этот механизм помогает разбить излишне сложный макет или функцию на составляющие и изолировать сложность от родительского документа. Для включения частичного представления используется синтаксис, уже встречавшийся ранее:

```
@Html.Partial("_Footer")
```

Частичное представление — всего лишь фрагмент представления. Его файл имеет то же расширение `cshhtml`, что и обычные представления, а для его поиска применяется та же стратегия, что и для поиска представлений. Хотя это и не обязательно, многие разработчики предпочитают снабжать имена частичных представлений префиксом `_` (подчеркивание), чтобы их было проще находить в папках представлений.

Частичное представление, как и полное, может быть сильно типизировано с определением модели; механика идентична представлениям, которые мы видели ранее. Однако при работе с частичным представлением контроллер не задействован, так что модель должна передаваться частичному представлению от представления.

```
@Html.Partial("_VolumeBarChart", Model.DailyRiderVolumeData)
```

Этот метод использования частичных представлений и моделей представлений бывает исключительно полезным в ситуациях с использованием составной

модели представления для построения чего-то вроде информационной панели с отчетами.

ПРИМЕЧАНИЕ

Не усложняйте частичные представления. Если информация, которую вы хотите вывести при помощи частичного представления, требует обращения к сервисам или зависит от бизнес-логики, рассмотрите возможность использования компонентов представления (эта тема рассматривается в главе 19 «Многоцветный код»).

Принцип работы частичных представлений несколько отличается от других средств работы с представлениями в Razor. Первое отличие: частичные представления не участвуют в секциях, определенных в макете, поэтому вы не сможете определить сценарную секцию для включения кода JavaScript, относящегося к частичному представлению в странице. Второе отличие заключается в том, что при визуализации частичное представление получает *копию* данных представления, а не ссылку на них, поэтому частичное представление не может изменить или задать значения в основном словаре данных представления. Следовательно, вы не сможете задать текст заголовка страницы из частичного представления через связь, определенную в вашем шаблоне. Наконец, частичные представления не визуализируются внутри макета, потому что предполагается, что они являются частью представления, которое уже решает эту задачу. Эти решения были приняты сознательно, чтобы частичные представления были простыми, прямолинейными и быстрыми в визуализации.

Использование расширенной функциональности Razor в представлениях

Вероятно, вы уже поняли, что Razor — не узкоспециализированный механизм представлений с ограниченной функциональностью. В ASP.NET Core MVC присутствуют новые замечательные возможности, которые дают разработчикам еще большую гибкость в отношении построения представлений и добавляют новые возможности в инструментарий разработчика.

Внедрение сервисов в представления

Когда разработчики услышали, что в механизм представлений будет встроена некая форма внедрения зависимостей, поднялась волна протестов, народ подписывал петиции, а в штате Вашингтон было подано несколько судебных исков.

Честно говоря, мы слегка преувеличили, но многие люди с полными основаниями беспокоились, что разработчики заходят на территорию, где в представления проникает бизнес-логика и обращения к сервисам, а представления начинают заниматься тем, чем должны заниматься контроллеры и модели. В конце концов, какой прок во фреймворке, который разделяет модели, контроллеры и представления, если все они начинают делать одно и то же?

Но если остановиться и немного подумать, вы поймете, что внедрение сервисов и так всегда происходило «за кулисами». Такие конструкции, как HTML-хелперы и URL-хелперы, доступны для вас и готовы выполнять такие операции, как разрешение маршрутов действий и контроллеров. И снова целью является достижение баланса и поиск инструмента, подходящего для поставленной задачи. Дело не в замене моделей представлений или контроллеров, а в расширении инструментария для ограничения работы, которая должна выполняться для каждого контроллера. Также полезно принять меры к тому, чтобы сервисы создавались в одном месте, могли нормально тестироваться и надежно использоваться в представлениях.

Вы можете использовать любые сервисы, зарегистрированные в контейнере сервисов в представлении — либо при помощи комбинации команды `using` и команды `inject`, либо при помощи команды `inject` с полным именем сервиса.

```
@using AlpineSkiHouse.Services
@inject ICsrInformationService customerServiceInfo
```

Здесь мы внедряем экземпляр `ICsrInformationService` и присваиваем ему имя `customerServiceInfo`. После этого переменная `customerServiceInfo` может использоваться в любой точке представления.

```
Call us at @customerServiceInfo.CallCenterPhoneNumber today!
```

ПРИМЕЧАНИЕ

За дополнительной информацией о включении сервисов в контейнеры обращайтесь к главе 14 «Внедрение зависимостей».

Впрочем, внедрение сервисов может использоваться неправильно. Например, частичному представлению может потребоваться информация о другом аспекте сеанса пользователя — скажем, о профиле или текущем содержимом покупательской корзины. Модель представления не следует загрязнять информацией такого рода, а размещать эту логику в каждом контроллере было бы неудобно. С другой стороны, было бы чудовищно неправильно внедрять в представление контекст базы данных, чтобы извлекать записи «на лету». Внедряя сервис, который может

получать данные, вы обеспечиваете хорошее разделение обязанностей, можете изолировать и тестировать сервисы по отдельности, и вам не приходится открывать контроллерам или моделям представлений доступ к информации, не связанной с непосредственно выполняемой задачей.

Работа с тег-хелперами

В новейшей версии Razor появилась мощная конструкция «тег-хелпер», позволяющая организовать обработку на стороне сервера без загрязнения HTML. Возможность описания обработки на стороне сервера при помощи элементов, напоминающих разметку HTML, появилась еще до Razor; так, в листинге 11.7 приведен фрагмент кода из проекта WebForms. В этом примере веб-элементы управления заменяют теги HTML `<input>`, `<label>` и ``.

Листинг 11.7. Использование веб-элементов управления в WebForms для построения обязательного поля

```
<div class="form-group">
  <asp:Label runat="server" AssociatedControlID="Email" CssClass="col-md-2
    control-label">Email</asp:Label>
  <div class="col-md-10">
    <asp:TextBox runat="server" ID="Email" CssClass="form-control"
      TextMode="Email" />
    <asp:RequiredFieldValidator runat="server" ControlToValidate="Email"
      CssClass="text-danger" ErrorMessage="The email field is required." />
  </div>
</div>
```

В WebForms каждый элемент управления должен помечаться соответствующим префиксом. Для большинства элементов управления, предоставляемых Microsoft, по умолчанию используется префикс `asp:`, но сторонние разработчики могут создавать собственные префиксы. Вы либо создаете собственный префикс, либо изменяете префикс и переводите его на нужное пространство имен. Также в каждый элемент управления необходимо включить атрибут `runat="server"` для обеспечения необходимой обработки.

Но есть один факт, который сбивает с толку дизайнеров, затрудняет изучение ASP.NET новичками и становится источником огорчений для многих — эти заполнители всего лишь занимают место и не содержат фактической разметки HTML. Возможности настройки страницы сильно ограничены, и каждый эскиз, поступающий от дизайнера, должен проходить серьезную реструктуризацию для включения функциональности на стороне сервера. Даже атрибут, используемый

для назначения классов CSS для элемента, отличается от соответствующих действительных конструкций HTML. Вы должны отложить в сторону свои знания общеизвестного синтаксиса и выбрать нечто такое, что не будет работать с другими технологиями.

А теперь сравните с кодом в листинге 11.8, где Razor используется с тег-хелперами. Синтаксис здесь намного более гибкий, код представляет собой действительную разметку HTML даже с нестандартными атрибутами, и вы четко видите, какие элементы при этом используются. Вам не нужно изучать, какие компоненты превратятся в `<label>` или ``, и вы снова можете использовать атрибуты элементов — например, для назначения класса CSS.

Листинг 11.8. Использование тег-хелперов в Razor для построения обязательного поля

```
<div class="form-group">
  <label asp-for="Email" class="col-md-2 control-label"></label>
  <div class="col-md-10">
    <input asp-for="Email" class="form-control" />
    <span asp-validation-for="Email" class="text-danger"></span>
  </div>
</div>
```

Здесь Razor демонстрирует свою мощь, и здесь заслуживают внимания несколько моментов. Во-первых, никакие атрибуты тег-хелпера не передаются клиенту в исходном виде. Такие конструкции, как `asp-for`, формируют контекст для тег-хелпера и помогают разработчику. Этот нестандартный атрибут может взаимодействовать с Visual Studio для предоставления поддержки IntelliSense (рис. 11.3). В этом случае тег-хелпер `label` располагает информацией о модели представления и программируется для получения свойства модели. Это позволяет выдавать рекомендации, основанные на типе модели; открытые свойства доступны для синтаксиса «привязки» такого рода. Достаточно высокий уровень интеграции IntelliSense позволяет распознавать `enum`-свойства и предлагать подходящие варианты автозавершения.

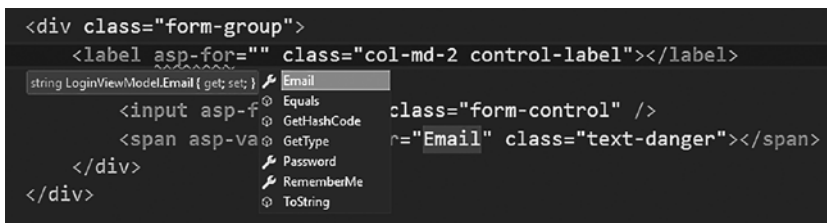


Рис. 11.3. Поддержка IntelliSense для тег-хелперов Razor в Visual Studio

После того как выполнение на стороне сервера будет завершено, окончательным результатом кода из листинга 11.8 становится код, приведенный в листинге 11.9. Все атрибуты тег-хелпера удалены, а все элементы HTML помечены атрибутами, обеспечивающими необходимую пользовательскую функциональность. Элемент `<input>` помечается сообщениями проверки на стороне клиента, а элементы `<label>` и `` настраиваются с учетом их отношений с `<input>`.

Листинг 11.9. Разметка HTML, сгенерированная для формы, созданной с использованием тег-хелперов

```
<div class="form-group">
  <label class="col-md-2 control-label" for="Email">Email</label>
  <div class="col-md-10">
    <input class="form-control" type="email" data-val="true" data-val-
email="The Email field is not a valid e-mail address." data-val-required="The Email
field is required."
id="Email" name="Email" value="">
    <span class="text-danger field-validation-valid" data-valmsg-for= "Email"
data-valmsg-replace="true"></span>
  </div>
</div>
```

Если неявное использование тег-хелперов вас не устраивает, также можно использовать более явно выраженный синтаксис. От новичков часто приходится слышать: «Как моя группа разработки поймет, какой код является частью HTML, а какой является частью тег-хелпера?» Обсуждение на эту тему, открытое с полгода назад, получило более 200 комментариев со стороны сообщества.

СОВЕТ

Подход команды ASP.NET к обсуждению технологии тег-хелперов и взаимодействию с сообществом при доведении ее до нормального уровня — превосходный пример ее серьезного отношения к концепции открытого кода. Разработчикам предлагается принять участие в обсуждении и вносить свой вклад в кодовую базу по адресу <https://github.com/aspnet/mvc>.

Чтобы разрешить использование явного префикса, добавьте следующую строку кода в свое представление.

```
@tagHelperPrefix "th:"
```

Этот синтаксис также можно включить на уровне `_ViewImports`; в этом случае все представления должны использовать одинаковую запись. Все элементы HTML, с которыми вы хотите использовать тег-хелперы, теперь должны снабжаться пре-

фиксом. В листинге 11.10 приведен код из листинга 11.8 с расставленными префиксами.

Листинг 11.10. Элементы формы, использующие тег-хелперы, с префиксами

```
<div class="form-group">
  <th:label asp-for="Email" class="col-md-2 control-label"></label>
  <div class="col-md-10">
    <th:input asp-for="Email" class="form-control" />
    <th:span asp-validation-for="Email" class="text-danger"></span>
  </div>
</div>
```

В приведенных примерах тег-хелперы демонстрируют некоторые преимущества перед веб-элементами управления: они компактны, не лишают разработчика свободы выбора и просто взаимодействуют с теми тегами, на которые они влияют. Если вы предпочитаете более обширный синтаксис с префиксами, вы сможете добавить его позже. Тег-хелперы имеют ограниченную область действия и могут объединяться в одном элементе для достижения динамического результата без риска для других частей страницы.

Существует еще один вариант применения тег-хелперов, в котором элементы HTML являются не целью, а артефактом использования тег-хелпера. Для примера возьмем тег-хелпер `environment` в листинге 11.11, который используется для генерации элемента `<link>` в соответствии с текущим окружением. В приведенном примере основной функцией тег-хелпера является простота получения нужных данных из представления. Нам не нужно знать, как получить доступ к конфигурации или написать логику в представлении для определения того, какая таблица стилей должна использоваться.

Листинг 11.11. Использование тег-хелпера `environment` для выбора таблицы стилей

```
<environment names="Development">
  <link rel="stylesheet" href="~/css/site.css" />
</environment>
<environment names="Staging,Production">
  <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
</environment>
```

При правильном использовании тег-хелперы абстрагируют сложности представления от разработчика. Они знают свой контекст и имеют доступ к модели представления, что существенно расширяет их возможности. Кроме того, вы можете строить собственные тег-хелперы. Если вы захотите узнать, как это делается, обращайтесь к главе 19.

Помощь со стороны Visual Studio

Visual Studio помогает разработчику понять, какие части его представления используют тег-хелперы. На рис. 11.4 показаны различия между тегами, которые не были помечены как тег-хелперы, и теми, которые были.

```
<div class="form-group">
  <label asp-for="CardHolderFirstName" class="col-md-2 control-label"></label>
  <div class="col-md-10">
    <input asp-for="CardHolderFirstName" class="form-control" />
    <span asp-validation-for="CardHolderFirstName" class="text-danger"></span>
  </div>
</div>
```

Рис. 11.4. Подсветка синтаксиса тег-хелпера в Visual Studio

В темной теме Visual Studio внешний элемент `<div>` на рис. 11.4 не изменяется тег-хелпером, но `<label>` внутри него изменяется. Элемент `<div>` окрашен в синий цвет, а `<label>` — в зеленый. Атрибуты HTML (например, `class` в `<div>`) окрашиваются в светло-голубой оттенок, а `asp-for` в `<label>` распознается как серверный атрибут тег-хелпера. Атрибут `asp-for` называется привязанным атрибутом и не включается в итоговый вывод для клиента, тогда как не привязанные атрибуты (такие, как `class`) просто переходят в HTML.

Предотвращение дублирования в представлениях

В вашем проекте наверняка будут пространства имен с высокой степенью повторного использования. Вместо того чтобы добавлять команды `using` в начало каждого представления, вы можете воспользоваться файлом `_ViewImports` для включения таких пространств имен в каждое представление в вашем приложении. Это относится как к сервисам, которые внедряются в представления, так и к тег-хелперам. Ниже приведены команды из файла `_ViewImports` проекта Alpine Ski House; файл находится в корне папки `Views`, чтобы модели и тег-хелперы, созданные командой, могли использоваться во всех представлениях:

```
@using AlpineSkiHouse
@using AlpineSkiHouse.Models
@addTagHelper *, AlpineSkiHouse.Web
```

`_ViewImports` следует той же модели наследования, что и другие компоненты механизма представлений. Все представления в папке совместно используют команды, содержащиеся в нем. Если вы захотите переопределить настройки из роди-

тельской папки, добавьте другой файл `_ViewImports` в дочернюю папку. Также можно переопределять определения сервисов.

Использование альтернативных механизмов представлений

Прежде чем завершать обсуждение Razor, следует сказать, что никто не заставляет вас использовать Razor. Вы можете написать собственный механизм представлений, реализовав `IViewEngine` (если вас не испугает такая перспектива), или же взять механизм представлений из другого фреймворка и использовать его вместо Razor.

```
services.AddMvc().AddViewOptions(options =>
{
    options.ViewEngines.Clear();
    options.ViewEngines.Add(MyIViewEngine);
});
```

На заре существования ASP.NET MVC было несколько альтернативных механизмов, которые получили некоторое распространение. Впрочем, большинство из них отошло на второй план с появлением Razor.

Конечно, написание механизма представления — нетривиальная задача. Возможно, вы не собираетесь делать все с нуля, а хотите унаследовать от существующего класса `RazorViewEngine` и внести несколько мелких поправок. Если вы выбрали такой путь, то сможете включить свою версию Razor в конфигурацию запуска, добавив ее в метод `ConfigureServices` класса `Startup`:

```
services.AddSingleton<IRazorViewEngine, YourCustomRazorEngine>();
```

И наконец, если Razor не делает чего-то, что вам нужно (например, если вы хотите расширить список провайдеров файлов или изменить каталоги, включаемые по умолчанию при поиске представлений), можно пойти по еще более простому пути и просто изменить некоторые настройки Razor по умолчанию:

```
services.Configure<RazorViewEngineOptions>(options =>
{
    // Изменение настроек
});
```

Итоги

Механизм представлений Razor уже доказал свою состоятельность как качественный, хорошо проработанный инструмент для обработки различных аспектов ви-

зуализации на стороне сервера. Переключение между разметкой HTML и кодом C# проходит беспрепятственно, а синтаксис позволяет легко включать различные функции фреймворка и языка. Синтаксис, который близко воспроизводит базовый синтаксис HTML, намного удобнее для дизайнеров, чем фреймворки, радикально отклоняющиеся от известного синтаксиса. С макетами и частичными представлениями создание веб-приложения, на страницах которого повторяются стандартные элементы, становится вполне реальным делом, тогда как модели с сильной типизацией, внедрение сервисов и тег-хелперов предоставляют вашим представлениям всю мощь платформы.

В книге несколько раз рассматривались вопросы изменения конфигурации, в том числе и в этой главе — применительно к механизму представлений Razor. Пришло время поближе присмотреться к этой теме. В следующей главе мы займемся конфигурацией и журналированием — двумя областями ASP.NET, в которых наблюдались значительные изменения и усовершенствования.

12

Конфигурация и журналирование

«Все понял. Спасибо за предупреждение, Чепмен. Нашей работе это не поможет, но, наверное, и не убьет. — Тим подождал, пока Грег завершит звонок, и в сердцах бросил телефонную трубку. — Вот черт!»

Грег Чепмен был менеджером по партнерской интеграции в телефонной компании и текущим владельцем системы телефонии и организации обслуживания клиентов, которая использовалась в Alpine. К сожалению, версия продукта, установленная в Alpine, не поддерживала прямое обновление без покупки дополнительного оборудования; а это означало, что команде разработчиков из Alpine не удастся воспользоваться сервисом, для которого они рассчитывали написать программу в целях отображения информации из центра обслуживания на сайте.

Два Марка проходили по коридору за дверью офиса Тима, когда до них донеслась завершающая реплика. «У нас вопрос, — начал Марк-1, просунув голову в дверь, — когда ты заканчиваешь разговор фразой “Вот черт!” и швыряешь трубку, нам уже стоит начинать беспокоиться за нашу работу?»

«Гмм, — проворчал Тим, — заходите». Приглашение запоздало, потому что Марки уже успели зайти. «Грег — мой старый приятель со времен работы в A.Datum, поэтому я обратился к нему, чтобы узнать насчет обновления программного обеспечения центра обслуживания. Он проверил наш номер лицензии, чтобы узнать, нельзя ли ускорить обработку запроса, но она безнадежно устарела. Мы даже не сможем обновить систему, пока не установим новые сервера, а это по сути означает...»

«...Что обещанного нового API не будет, — закончил за него Марк-1. — Прекрасно».

«Да. Я знаю, что вам это было нужно для сайта, — продолжал Тим, — но я не думаю, что мы получим его до сдачи сайта. Придется обходиться тем, что есть».

«Ну это же не катастрофа, верно? — спросил Марк-2. — Я имею в виду — система записывает файл с информацией каждый раз, когда кто-нибудь подключается по VOIP-телефону. Я знаю, что это было нужно для старых отчетов об эксплуатации, но мы можем просто приказать системе каждый раз перезаписывать файл. Помнишь, Даниэль рассказывала о настраиваемых провайдерах конфигурации?»

«Точно... с этого и начнем! — Марк-2 оживился от мысли о том, чтобы повозиться с новой технологией и показать ее Даниэль. — Тим, постой. Мы немного поработаем с этим на пару и определим, насколько серьезна проблема. Дай нам три часа — посмотрим, что из этого выйдет».

Журналы и конфигурация играют исключительно важную роль в эксплуатации продукта. В этой главе рассматривается новая система конфигурирования ASP.NET Core, а также новые мощные инструменты журналирования, которые упрощают создание, сохранение и передачу информации о вашем приложении и ходе его выполнения.

Расставание с web.config

Традиционно вся конфигурация приложения хранилась в файле `web.config`; это было и роскошью, и бременем. Со временем файл `web.config` стал вмещать все более широкие возможности, но его специализация стала размываться. При желании в нем можно подключить библиотеки, предоставить доступ конкретному пользователю, изменить настройки механизма представлений, задать строки подключений и установить цвет фона сайта по умолчанию — слишком много всего для одного файла.

Сложность стала одним из движущих факторов отказа от `web.config`. Имя файла намекает, что в файле хранится какая-то конфигурация, которая имеет отношение к «веб»... но к чему именно? Веб-приложению? Веб-серверу? Еще чему-нибудь?

Сообщество .NET привыкло к некоторым правилам, которые выполняются большинством приложений. У этих правил есть несколько исключений. Сначала приложение разрабатывается в Visual Studio. На оборудовании, на котором развертывается рабочее приложение, должна работать система Windows Server, а на сервере должна быть установлена платформа .NET Framework. Также нужен экземпляр IIS для размещения веб-сайта и соответствующей конфигурации хоста. Вероятно, нужно будет написать код для взаимодействия с установкой SQL Server, чтобы хранить данные приложения. Поскольку все эти предположения с ASP.NET Core уходят на второй план, файл `web.config` должен уйти вслед за ними.

Последнее утверждение истинно по крайней мере с точки зрения приложения. Если вы собираетесь разместить свое приложение в IIS, без файла `web.config` не обойтись, но управлять конфигурацией вашего приложения в ASP.NET Core будет не он. Освобождение от `web.config` означает, что инструментарий, используемый вами, не обязан поддерживать `web.config`. Если вы хотите разместить свое приложение вне IIS, вероятно, встроенная поддержка `web.config` использоваться не будет, потому что хост ее не поймет. И если уж откровенно, в большинстве фай-

лов `web.config` содержится столько разнородной информации и параметров конфигурации, что от них в ужасе убежит стая рогатых монстров.

И все равно один из самых серьезных и раздражающих недостатков `web.config` заключается в том, что им часто злоупотребляют. Унаследованная общесистемная конфигурация из папок IIS более высокого уровня (`applicationhost.config` или `machine.config`) может непреднамеренно проникнуть в ваше приложение, усложняя его диагностику. Настройки родительских уровней могут добавлять или фиксировать аспекты поведения или провайдеров, о которых вы не подозреваете и которые не существуют в других средах, создавая нежелательные побочные эффекты во время выполнения.

В ASP.NET Core вы заметите, что тип конфигурации для загрузки можно выбрать произвольно, что конфигурации хоста и сервера могут отделяться от настроек приложения, а вы можете использовать в своем проекте более мощную систему конфигурирования.

Настройка конфигурации приложения

Каждая созданная вами среда обладает собственными настройками, соответствующими ее контексту. При запуске на локальной машине разработки для Azure Storage Tools можно использовать эмулятор разработчика, но в средах более высокого уровня используются реальные инструменты.

Использование `ConfigurationBuilder` четко приказывает вашему приложению прочесть эти значения из источника, определяемого средой, в конструкторе класса `Startup`:

```
var builder = new ConfigurationBuilder()
    .SetBasePath(env.ContentRootPath)
    .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
    .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true);

if (env.IsDevelopment())
    builder.AddUserSecrets();

builder.AddEnvironmentVariables();
Configuration = builder.Build();
```

Вероятно, из этого кода вы уже сделали вывод, что среда, в которой выполняется приложение, оказывает влияние на используемую конфигурацию. Вы можете использовать маркер `env.EnvironmentName` для применения файла конфигурации в зависимости от среды. Маркер берется из переменной среды средствами операционной системы либо, в том случае, если вы используете Visual Studio, — из переменных, связанных с процессом, задаваемых в отладочной конфигурации проекта. Механизм условного добавления источников конфигурации позволяет легко заменять значения без дополнительных шагов сборки или развертывания. У вызова

.AddJsonFile, основанного на значениях среды, параметр `optional` равен `true`; это означает, что эти файлы должны создаваться только тогда, когда это необходимо. Кроме того, из-за особенностей хранения значений конфигурации вам не нужно беспокоиться о создании всех значений конфигурации для всех вариантов среды. Например, в файле среды разработки может определяться строка подключения, которая не существует в других конфигурациях.

ПРИМЕЧАНИЕ

Некоторые типы данных не должны храниться в файлах конфигурации приложения, особенно если вы работаете с открытыми репозиториями. Даже при работе в команде существуют аспекты конфигурации, не предназначенные для общего доступа. О том, как хранить такие значения, вы узнаете в главе 13 «Идентификация, безопасность и управление правами», когда мы будем рассматривать скрытые данные пользователей.

Код также явно показывает, что некоторые значения могут храниться в нескольких файлах или даже запрашиваться от разных провайдеров конфигурации. Здесь есть вызовы для файлов JSON и переменных среды. Если код выполняется в среде разработки, он также загружает конфигурации из хранилища скрытых данных пользователей.

Порядок загрузки конфигураций важен, потому что провайдеры конфигурации вызываются в порядке их добавления. Значения, существующие в нескольких источниках конфигурации, заменяют предыдущие значения без предупреждения. Если каждый из файлов `appsettings.json` и `appsettings.development.json` содержит значение `StorageKey`, и этот же ключ также существует в переменной среды, то во время выполнения будет использоваться значение из переменной среды.

Все данные конфигурации хранятся в парах «ключ — значение», даже если вы захотите выразить иерархическую структуру при их сохранении. Обычно вы видите наборы значений вроде тех, которые включаются в файл `appsettings.json`:

```
"Logging": {  
  "IncludeScopes": false,  
  "LogLevel": {  
    "Default": "Debug",  
    "System": "Information",  
    "Microsoft": "Information"  
  }  
}
```

Еще раз подчеркнем: данные конфигурации могут храниться только в парах «ключ — значение», поэтому приведенная выше конфигурация преобразуется в форму, подходящую для такой структуры. Если вы установите точку прерывания в конце конструктора `Startup` и просмотрите свойство `Configuration`, то увидите, что значения были преобразованы в соответствующие значения из табл. 12.1.

Таблица 12.1. Иерархические ключи из конфигурации и их соответствие уровням записи в журнал

Ключ	Значение
Logging:IncludeScopes	False
Logging:LogLevel:Default	Debug
Logging:LogLevel:System	Information
Logging:LogLevel:Microsoft	Information

Провайдер конфигурации проходит дерево значений в документе JSON до конечного узла, где он извлекает значение. Для всех остальных имен свойств на пути он использует имена этих узлов для построения соответствующего ключа.

Использование готовых провайдеров конфигурации

Появляется все больше готовых компонентов, предоставляемых фреймворками для работы с конфигурацией, а сообщество активно выступает за поддержку дополнительных типов хранения. Все серьезные провайдеры дополняются методами расширения для `ConfigurationBuilder`, которые упрощают подключение к источникам и применение цепочек в динамическом синтаксисе.

Ниже перечислены все доступные провайдеры из репозитория конфигурации ASP.NET на GitHub с примерами использования.

- `Configuration.Azure.KeyVault` — читает данные из заданного хранилища ключей для заданного идентификатора клиента и скрытых данных. Этот механизм позволяет разработчикам использовать облачные ресурсы с детальным управлением ресурсами, которые им доступны.

```
builder.AddAzureKeyVault(vaultUri, clientId, clientSecret);
```

- `Configuration.CommandLine` — позволяет передавать аргументы из командной строки. Вероятность использования этого провайдера в приложениях ASP.NET Core MVC невелика, но это возможно при выполнении нескольких дополнительных шагов.

```
builder.AddCommandLine(args);
```

- `Configuration.EnvironmentVariables` — данные конфигурации читаются из процесса, в котором выполняется приложение. Помните, что при запуске процесса переменные среды (пользовательские и системные) загружаются без обновления, поэтому процессы и подпроцессы Visual Studio не заметят изменений, вносимых в настройки. Если вы хотите работать с измененными значениями, подумайте о задании переменных среды на вкладке **Debug** страниц свойств ва-

шего проекта. Также можно отфильтровать переменные среды по префиксу, извлеченному из имени ключа.

```
builder.AddEnvironmentVariables("AlpineApi-");
```

- **Configuration.Ini** — хотя об INI-файлах редко упоминают в контексте веб-приложений, они неплохо подходят для хранения конфигурации. INI-файлы поддерживают именованные секции, в основном свободны от лишнего «балласта» и нормально читаются людьми.

```
builder.AddIniFile("loadbalancerinfo.ini", optional: false, reloadOnChange: true);
```

- **Configuration.Json** — провайдер, фактически используемый в шаблоне проекта ASP.NET Core MVC. JSON — не самый понятный формат, но файлы JSON предоставляют поддержку вложенной структуры, а документы с данными в формате JSON могут проверяться на синтаксическую правильность. Существует много программ «красивой печати», которые позволяют легко произвести визуальную оценку документа и понять его содержимое.

```
builder.AddJsonFile("appsettings.json", optional: true, reloadOnChange: true);
```

- **Configuration.Xml** — традиционный формат хранения данных ASP.NET, знакомый большинству разработчиков с опытом программирования .NET. Кроме того, если прежде вы работали с нестандартными секциями конфигурации и хотите перенести их, портирование этих настроек в проект ASP.NET Core MVC потребует минимальных усилий. Провайдер конфигурации XML позволит вам просмотреть как атрибуты, так и значения узлов из разметки XML.

```
builder.AddJsonFile("webserviceendpoints.{envEnvironmentName}.json", optional: true);
```

Файловые провайдеры для форматов XML, INI и JSON поддерживают необязательное использование файлов конфигурации и автоматическую перезагрузку при изменениях во время выполнения без необходимости перезапуска приложения. Чтобы включить эту возможность, присвойте параметру `reloadOnChange` значение `true` при добавлении этих типов файлов конфигурации; в этом случае изменения станут немедленно видны вашему приложению.

Построение собственного провайдера конфигурации

Когда два Марка решили взяться за проблему доступности представителя отдела обслуживания клиентов на сайте, они нацелились на файл, содержащий необходимую информацию, и решили рассматривать его как источник конфигурационных данных. Файл содержит строку заголовка с телефонным номером контактного центра, дополнительную строку для каждого представителя, который находится

в сети, дополнительный номер для связи с ним и время начала работы. Они взяли пример файла для анализа информации:

```
ACTIVE_PHONE_NUMBER|555-SKI-HARD
STAMMLER, JEFF|4540|2016-09-12T08:05:17+00:00
ILYINA, JULIA|4542|2016-09-12T13:01:22+00:00
SCHUSTIN, SUSANNE|4548|2016-09-12T12:58:49+00:00
CARO, FERNANDO|4549|2016-09-12T10:00:36+00:00
```

А теперь посмотрим, как создать поддержку такого файла в приложении. Сначала нужно написать класс, который разбирает файл, берет информацию и собирает значения конфигурации. Это важно, потому что эту логику и функциональность можно спроектировать и протестировать отдельно от других проблем. В действительности нас интересуют два конкретных значения: телефонный номер контакт-центра и количество работающих операторов. Извлеките и вычислите эти значения, затем сохраните их в словаре с ключом, имеющим префикс `CsrInformationOptions`. Следующий метод `Parse` читает поток и выполняет основную часть работы:

```
public IDictionary<string, string> Parse(Stream stream)
{
    _data.Clear();

    using (var reader = new StreamReader(stream))
    {
        // Первая строка содержит номер телефона
        var line = reader.ReadLine();
        var phoneNumber = ExtractPhoneNumber(line);

        // Во всех остальных строках содержатся данные контактов
        var onlineCount = 0;
        while ((line = reader.ReadLine()) != null)
        {
            onlineCount++;
        }

        // Добавление в хранилище конфигурации
        _data.Add("CsrInformationOptions:PhoneNumber", phoneNumber);
        _data.Add("CsrInformationOptions:OnlineRepresentatives", onlineCount.
            ToString());
    }

    return _data;
}
```

Реализация нестандартного провайдера конфигурации требует написания реализации `IConfigurationSource`, отвечающей за настройку и возвращение экземпляра `IConfigurationProvider`. Для конфигурации на базе файла можно воспользовать-

ся встроенным источником `FileConfigurationSource`, который содержит свойства, необходимые для описания файла, и предоставляет возможность автоматической перезагрузки при изменении файлов на диске.

```
public class CsrInformationConfigurationSource : FileConfigurationSource
{
    public override IConfigurationProvider Build(IConfigurationBuilder builder)
    {
        FileProvider = FileProvider ?? builder.GetFileProvider();
        return new CsrInformationConfigurationProvider(this);
    }
}
```

Провайдер предоставляет набор методов для обращения к внутреннему списку значений. Система конфигурации использует эти методы для обращения к значениям, прочитанным из источника конфигурации. Здесь мы реализуем класс `CsrInformationConfigurationProvider`, который наследует от `FileConfigurationProvider` и использует построенный ранее парсер для чтения данных из переданного потока.

```
public class CsrInformationConfigurationProvider : FileConfigurationProvider
{
    public CsrInformationConfigurationProvider(FileConfigurationSource source) :
base(source) {

        public override void Load(Stream stream)
        {
            var parser = new CsrInformationParser();
            Data = parser.Parse(stream);
        }
}
```

Хотя это и не обязательно, также можно явно обозначить использование источника конфигурации через метод расширения, добавляемый в `IConfigurationBuilder`. Это упрощает сцепление источника конфигурации с другими источниками конфигурации в начале работы. Класс `CsrInformationExtensions` содержит эти методы расширения в проекте *Alpine* и в конечном итоге позволяет добавить файл конфигурации в проект в `Startup.cs` всего одной строкой кода:

```
var builder = new ConfigurationBuilder()
    .SetBasePath(env.ContentRootPath)
    .AddCsrInformationFile("config\\TPC_ONLINE_AGENTS.INFO", reloadOnChange: true)
    .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
    .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true);
```

Впрочем, это всего лишь одно из возможных решений, которое может не подойти для вашей конкретной ситуации. Всегда анализируйте, какие аспекты ввода ваше-

го приложения должны считаться источниками данных, а какие — источниками конфигурации.

ПРИМЕЧАНИЕ

На момент написания книги провайдеры `FileConfigurationSource` вопреки замыслу разработчиков не перезагружались при изменении данных на диске. Это известная проблема версии 1.0.1, а команда ASP.NET установила на GitHub контрольную точку разрешения этой проблемы в версии 1.1.0.

И последнее напоминание: хотя мы создали нестандартного провайдера конфигурации для извлечения предельно конкретного набора значений из файла с уникальным форматом, в итоге свойства были преобразованы в пары «ключ — значение» и сохранены в общем наборе. Это означает, что любой источник конфигурации, добавленный в приложение после нашего, может содержать свое значение для наших ключей.

Применение опций

Целью применения механизма конфигурирования в ASP.NET Core является упрощение поставки и потребления конфигурационных данных. JSON, XML, а также другие нестандартные источники данных обращаются к поставщику данных, а опции (options) упрощают потребление. Опции позволяют использовать внедрение зависимостей для передачи конфигурационных данных нашим сервисам, контроллерам и другим объектам приложения.

Чтобы в полной мере использовать эту возможность, нужен объект для внедрения, поэтому начнем именно с него. Вспомните, что конфигурационная информация в нашем примере с обслуживанием клиентов по сути сводится к двум свойствам: телефону контакт-центра и количеству операторов, работающих в настоящее время. По этим данным также можно сделать вывод о том, открыт ли контакт-центр. Объекты опций представляют собой обычные объекты CLR; это означает, что создание класса опций — задача достаточно прямолинейная.

```
public class CsrInformationOptions
{
    public string PhoneNumber { get; set; }
    public int OnlineRepresentatives { get; set; }
}
```

Мы хотим предоставить эти данные сервисам опций, доступным на протяжении жизненного цикла приложения, и поэтому вернемся к классу `Startup` в методе `ConfigureServices` для необходимого связывания. Вызов `Configuration.GetSection` позволяет получить группу данных с общим префиксом имен. Это может быть секция файла конфигурации или — как в нашем случае — значения,

добавленные в хранилище конфигурации с префиксом `CsrInformationOptions`. Результат вызова передается методу `Configure<T>` и приводится к типу класса опций, созданному ранее.

```
services.Configure<CsrInformationOptions>(Configuration.GetSection  
    ("CsrInformationOptions"));
```

При выполнении приложения появляется возможность запросить объект конфигурации, все свойства которого заранее заполняются значениями, загруженными из различных источников конфигурации. Пример такого рода встречается в конструкторе `CsrInformationService`.

```
public CsrInformationService(IOptions<CsrInformationOptions> options)  
{  
    _options = options.Value;  
}
```

Обратите внимание на передачу в конструкторе параметра типа `IOptions<CsrInformationOptions>` вместо простого `CsrInformationOptions`. Это связано с особенностями функционирования сервисов опций: в приложении регистрируется только один сервис, способный работать с любым сконфигурированным объектом опций.

Сервисы опций обычно подключаются по умолчанию на различных ветвях кода, используемых шаблоном проекта по умолчанию. Если вы получите ошибку с сообщением о том, что реализация `IOptions<TOptions>` в вашем приложении не настроена, проблема решается подключением вызова `services.AddOptions()` в метод `ConfigureServices` класса `Startup`.

Операции с журналом

Как хорошо, что мы ничего не сохраняли в журнале; это бы сильно затруднило диагностику возникшей проблемы.

Никто... и никогда

Какая бы неразбериха ни творилась в отношении соглашений, форматов и участия в практике журналирования, одна истина остается непреложной: невозможно начать запись в журнал «задним числом», чтобы найти причины уже случившейся ошибки. Если вы не сохранили данные в журнале, когда что-то произошло, вы тем самым упустили важнейшую возможность докопаться до истинных причин происходящего. Журнал — более честный источник информации, чем пользователь; более оперативный, чем подача заявки в службу поддержки; и даже более уравновешенный, чем встревоженный менеджер по интеграции разработки и эксплуатации. Без журнала мы почти ничего не знали бы о внутреннем состоянии

нашего приложения во время выполнения и лишились бы важного инструмента профилактического сопровождения кода.

В соответствии с этим принципом ASP.NET Core подняло уровень доступности информации для разработчиков на новую ступень. Им уже не нужно обращаться к сторонним библиотекам и извлекать конфигурационные файлы из примеров в интернете; теперь шаблон проекта включает все, что необходимо знать для простой организации журналирования в приложении. Взгляните на следующие поля и конструктор `CsrInformationService`:

```
private ILogger<CsrInformationService> _logger;
private CsrInformationOptions _options;

public CsrInformationService(
    IOptions<CsrInformationOptions> options,
    ILogger<CsrInformationService> logger)
{
    _options = options.Value;
    _logger = logger;
    logger.LogInformation("Entered the CsrInformationService constructor.");
}
```

Чтобы начать журналирование в приложении, вам даже не нужно создавать экземпляр `ILogger`; он предоставляется сервисами, доступными в результате внедрения зависимости, которое более подробно рассматривается в главе 14 «Внедрение зависимостей». Во время выполнения фабрика создает экземпляр `ILogger` с типом класса сервиса, который задает категорию записи в журнал, и мы можем использовать объект, как только он окажется в области видимости. Все, что нужно для журналирования, — это вспомогательное поле, параметр конструктора и одна строка кода.

Что еще важнее — все, что связано с журналированием, является абстракцией над концепциями журнальных данных. Эта концепция более подробно рассматривается ближе к концу главы, когда встроенный механизм журналирования будет заменен альтернативной библиотекой, что позволяет легко использовать выбранную вами библиотеку.

Создание журналов с доступной информацией

Если ситуация выходит из-под контроля, а у вас имеется журнал с понятной, четкой информацией, вы сможете обнаружить сбой в работе сервиса, проанализировать влияние изменений новой версии на производительность и понять, что вообще произошло. Идеальных журналов не бывает, но, если уж понадобится, лучше вооружиться информацией, которая обеспечит ранний успех.

Несколько советов по поводу того, как сохранять более полезную информацию в журналах ваших приложений:

- **Информация должна быть ясной:** вероятно, вы запомнили «пять главных вопросов» на уроках в школе: кто, что, где, почему и как. Во многих ситуациях существуют важные подробности, которые должны включаться в сообщения, генерируемые для файлов журналов. Информация о пользователе (более конкретная), идентификатор запроса (анонимный), выполняемая операция, сервис, время и меры, принятые для выхода из аварийной ситуации, — вся эта информация будет полезной.
- **Не переусердствуйте:** попробуйте предугадать, какая информация вам понадобится, и включайте ее в запросы, но также ограничивайтесь информацией, актуальной для вашей ситуации. Когда пользователь пытается обновить свой профиль, не нужно сохранять в журнале количество процессоров на сервере.
- **Не жадничайте:** дисковое пространство стоит дешево, а время, требуемое для анализа проблемы, драгоценно. Если все сделано правильно, подробная информация, относящаяся к текущему контексту, может сильно пригодиться в тяжелый момент. Временная информация, недоступная вне области видимости запроса, также может оказаться чрезвычайно ценной. Полезной будет любая информация, которая обеспечит группировку и получение информации о более широком контексте выполнения. Одно «но»: следите за тем, чтобы выводимая в журнал информация не была настолько обширной, чтобы это повлияло на производительность.
- **Знайте, какая информация у вас уже есть:** не нужно делать то, что уже сделано. Повысьте уровень детализации для журнальных данных зависимостей (см. следующий раздел) и посмотрите, какие возможности предоставляют библиотеки в стандартной функциональности записи в журнал. Если библиотека, которую вы используете, уже способна предоставить нужную информацию, постепенно включайте фильтры на уровне пространств имен и дайте им возможность постараться и выдать всю необходимую информацию.
- **Помните о возможных рисках:** сохраненная информация может сыграть ключевую роль в быстрой реакции на неисправность приложения, но вы должны осмотрительно выбирать данные для сохранения. Например, Сьюзен Шустин совершенно не нужно, чтобы кто-нибудь знал, что она живет по адресу 1744 North Cypress Blvd, Winnipeg, MB, R0M 2J4 и имеет канадский номер социального страхования 636 192 352. Пожалуйста, дорогой читатель, следите за своими манерами при записи в журнал. Было бы несправедливо делать кого-нибудь из разработчиков нарушителем закона о защите персональных данных (который ваша организация обязана соблюдать) только потому, что он открыл файл журнала.

Информация об исключениях

Советы по поводу того, какая информация должна сохраняться в журнале о возникающих исключениях, чрезвычайно разнообразны. Если исключение просто перезапускается, зачем вообще возиться с сохранением? Если у вас нет полной информации, как узнать, что пошло не так? Возможно, стоит взглянуть на исклю-

чения с другой точки зрения и учесть состояние приложения, которое привело к генерации исключения.

Обработка исключений происходит в исключительных обстоятельствах, когда у вас имеется критическая ветвь приложения и возможность что-то сделать с возникшей ошибкой. Не стоит обрабатывать исключения, после которых вы не собираетесь восстанавливать работоспособность (разве что они предоставляют особо ценную информацию о состоянии приложения).

Если в вашей ситуации операция, сопряженная с высоким риском, может предоставить критичную информацию, выполните рекомендации по сохранению ясной информации (см. выше). Уровни журналирования будут описаны ниже, а пока приведем пример стратегии, которая может использоваться для активной диагностики проблем во время выполнения:

```
try
{
    _logger.LogDebug($"Entering critical path for {resource.Id}...");
    _service.InvokeRiskyManeuvre();
    _logger.LogInformation("Successfully completed barrel roll for
                           {resource.Id}.");
}
catch (UnexpectedStateException ex)
{
    _logger.LogError($"Unable to complete operation, resource {resource.Id} in
                     {resource.Status}
state. Error: {ex.Message}");
    throw;
}
```

В разных конфигурациях журнала вы получите разные результаты, но во всех ситуациях необходимо знать, с каким ресурсом вы работаете. В отладочной конфигурации ход выполнения можно отслеживать по строкам. В средах более высокого уровня приложение можно настроить для вывода сообщений об успехе. Кроме того, при возникновении исключения вы явно перехватываете исключение, о котором вам известно, стараетесь изолировать ошибки, которые можно было предвидеть, а затем предоставляете информацию о ресурсе, которая поможет найти причины ошибки.

Журналирование как стратегия разработки

Итак, после обнаружения ошибки вы начинаете сохранять в журнале Всё Подряд. Конечно, почему бы и нет? Когда вы работаете на своей локальной машине, журнал может предоставить готовую информацию о том, какие сервисы работали в то время, когда приложение не находилось под нагрузкой, и порядок событий можно было достаточно легко проанализировать. Несколько примеров ситуаций, в которых журнал может заметно упростить отладку:

- Анализ того, что происходит в приложении при асинхронном возникновении событий.
- Проектирование сервиса, реагирующего на команды источника «Интернета вещей» (а не приложения).
- Разработка микросервиса, взаимодействующего со многими другими сервисами; локально вы можете заниматься отладкой одного сервиса и включить журналирование для сохранения периферийной информации других сервисов.
- При попытках локально повторить сложную ошибку, связанную с конфиденциальной информацией, вы можете сохранить данные и воспроизвести их через локальный поток журнальных данных.

Если вы поморщились при виде последнего пункта, скорее всего, вы упускаете полезную возможность. Локально вводом данных с большой вероятностью будете заниматься вы сами, а вся секретная информация вам и без того известна. Использование трассировочного журналирования в сочетании с записью более высокого уровня — отличный способ сохранения подробной информации о том, что происходило в вашем приложении.

```
_logger.LogDebug($"Created new resource {resource.Id}...");  
_logger.LogTrace($"Resource private token {resource.Token}");
```

Сохранение такой информации в продуктивной среде наверняка вызовет неодобрительные взгляды; к счастью, трассировочное журналирование по умолчанию отключено. ASP.NET Core позволяет задать уровень журналирования, что позволяет сократить объем лишних деталей в журнальных файлах приложений.

Уровни журналирования в ASP.NET Core

К этому моменту вы уже поняли наше неоднозначное отношение к размещению конфиденциальной информации в журналах. Вероятно, вы также заметили некоторые методы, которые использовались для журналирования; давайте посмотрим, как команда Alpine Ski Team воспользовалась ими. Все началось с самостоятельного определения уровней журналирования. Эти уровни перечислены ниже по порядку, от самых полезных в ходе разработки до самых содержательных в условиях эксплуатации продукта.

- **Trace**: этот уровень отключен по умолчанию. Используйте этот уровень локально для диагностики проблем с конфиденциальными данными. Так как среди разработчиков считается, что он аналогичен нулевому уровню безопасности, никогда не включайте его в рабочих средах.
- **Debug**: на этом уровне можно различать условия выбора стратегии, состояния переключателей и флагов и различные значения, представляющие интерес в основном для разработчика (например, GUID и идентификаторы процессов).

- **Information:** этот уровень особенно полезен для разработчиков, занимающихся программированием операций, а также при работе с данными, используемыми программами анализа журнальных данных для создания сигналов, уведомлений и других контрольных функций.
- **Warning:** этот уровень применяется, когда еще не все потеряно, но нет и полной ясности картины. Идеальная ситуация для уровня **Warning** — когда после возникновения ошибки работоспособность программы может быть восстановлена повторной попыткой на более высоком уровне, и надежда еще может остаться.
- **Error:** если вам потребуется отменить операцию, критическую для деятельности приложения, и сохранить в журнале информацию, жизненно важную для последующего анализа, используйте этот уровень. Ошибки могут относиться к конкретному запросу, сообщая о сбое в конкретных обстоятельствах.
- **Critical:** этот уровень предназначен для серьезных сбоев приложения — например, недоступности критической инфраструктуры или другой катастрофической ситуации. На тормоза жать поздно, машина уже в кювете.

А теперь посмотрим, как задействовать все эти уровни в приложении, а именно как настроить приложение для сохранения информации в журнале на нужном уровне. Вернитесь к файлу `appsettings.config`; конфигурация, заданная группой `Alpine`, уже приводилась ранее в этой главе.

```
"Logging": {  
  "IncludeScopes": false,  
  "LogLevel": {  
    "Default": "Debug",  
    "System": "Information",  
    "Microsoft": "Information"  
  }  
}
```

По умолчанию для всех операций с журналом используется уровень **Debug**, но также присутствуют записи для пространств имен **System** и **Microsoft**, в которых также задаются уровни записи в журнал. Группа решила, что эти настройки будут полезны для большинства разработчиков при локальном извлечении кода и запуске приложения. Результаты показаны на рис. 12.1: приложение запускается в консольном режиме командой `dotnet run`; однако вы видите, что выполнение типичных действий (например, переход от представления к представлению) сопровождается большим количеством информационного шума.

ПРИМЕЧАНИЕ

Этот шум — именно то, чего нам хотелось бы избежать, поэтому команда решила использовать скрытие данных, для того чтобы спрятать некоторые подробности на своих компьютерах. Скрытые данные пользователей более подробно рассматриваются в главе 13.

```

posh-git ~ AlpineSkiHouse [ftr/logging]
Request starting HTTP/1.1 GET http://localhost:5000/Home/Contact
info: AlpineSkiHouse.Services.CsrInformationService[0]
  Entered the CsrInformationService constructor.
info: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[1]
  Executing action method AlpineSkiHouse.Controllers.HomeController.Contact (AlpineSkiHouse.Web) with arguments () - ModelState is Valid
info: Microsoft.AspNetCore.Mvc.ViewFeatures.Internal.ViewResultExecutor[1]
  Executing ViewResult, running view at path /Views/Home/Contact.cshtml.
info: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[2]
  Executed action AlpineSkiHouse.Controllers.HomeController.Contact (AlpineSkiHouse.Web) in 88.2322ms
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[2]
  Request finished in 93.0733ms 200 text/html; charset=utf-8
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[1]
  Request starting HTTP/1.1 GET http://localhost:5000/
info: AlpineSkiHouse.Services.CsrInformationService[0]
  Entered the CsrInformationService constructor.
info: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[1]
  Executing action method AlpineSkiHouse.Controllers.HomeController.Index (AlpineSkiHouse.Web) with arguments () - ModelState is Valid
info: Microsoft.AspNetCore.Mvc.ViewFeatures.Internal.ViewResultExecutor[1]
  Executing ViewResult, running view at path /Views/Home/Index.cshtml.
info: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[2]
  Executed action AlpineSkiHouse.Controllers.HomeController.Index (AlpineSkiHouse.Web) in 7.4034ms
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[2]
  Request finished in 12.8406ms 200 text/html; charset=utf-8
dotnet.exe[64]:12668
  
```

Рис. 12.1. Избыток информации в конфигурации записи в журнал по умолчанию

Чтобы отфильтровать информационные сообщения в условиях реальной эксплуатации продукта, команда также добавила файл конфигурации `appsettings.production.json` с уровнем журналирования, который допустим для рабочей среды:

```

{
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Information",
      "System": "Error",
      "Microsoft": "Error"
    }
  }
}
  
```

Перед вами та же структура, что и в главном файле конфигурации. Вспомните, о чем говорилось ранее в этой главе: одни и те же значения могут встречаться в нескольких источниках конфигурации, но «побеждает» последнее значение. В данном случае переопределение помогает убрать информационные сообщения, являющиеся частью пространств имен `System` и `Microsoft`, и позволяет увидеть действительно нужную информацию (рис. 12.2).

```

posh-git ~ AlpineSkiHouse [ftr/logging]
C:\james\code\AlpineSkiHouse\src\AlpineSkiHouse.Web [ftr/logging +1 ~3 -0 !]> dotnet run
Project AlpineSkiHouse.Web (.NETCoreApp,Version=v1.0) was previously compiled. Skipping compilation.
Hosting environment: Development
Content root path: C:\james\code\AlpineSkiHouse\src\AlpineSkiHouse.Web
Now listening on: http://localhost:5000
Application started. Press Ctrl+C to shut down.
info: AlpineSkiHouse.Services.CsrInformationService[0]
  Entered the CsrInformationService constructor.
warn: AlpineSkiHouse.Controllers.HomeController[0]
  The application client secret key was less than the expected length.
fail: Microsoft.EntityFrameworkCore.Query.Internal.SqlServerQueryCompilationContextFactory[1]
  An exception occurred in the database while iterating the results of a query.
  System.Data.SqlClient.SqlException: Cannot open database "aspnet-AlpineSkiHouse-4c485a7e-b4ac-4816-8c-f5-4e2568b8f069" requested by the login. The login failed.
  Login failed for user 'ORANGEBUCKET\James'.
    at System.Data.SqlClient.SqlInternalConnectionTds..ctor(DbConnectionPoolIdentity identity, SqlConnectionString connectionOptions, Object providerInfo, Boolean redirectedUserInstance, SqlConnectionString userConnectionOptions, SessionData reconnectSessionData, Boolean applyTransientFaultHandling)
    at System.Data.SqlClient.SqlConnectionFactory.CreateConnection(DbConnectionOptions options, DbConnectionPoolKey poolKey, Object poolGroupProviderInfo, DbConnectionPool pool, DbConnection owningConnection, DbConnectionOptions userOptions)
    at System.Data.ProviderBase.DbConnectionFactory.CreatePooledConnection(DbConnectionPool pool, DbConnection owningObject, DbConnectionOptions options, DbConnectionPoolKey poolKey, DbConnectionOptions userOptions)
    at System.Data.ProviderBase.DbConnectionPool.CreateObject(DbConnection owningObject, DbConnectionOptions userOptions, DbConnectionInternal oldConnection)
dotnet.exe[64]:13204
    < 160529[64] 1/1 [-] NUM PRI: 107x26 (1,104) 25V 22076 100%

```

Рис. 12.2. Изменение конфигурации для устранения лишней информации в журнале

Чтобы перенести эти настройки в приложение, необходимо правильно настроить промежуточное ПО, передавая секцию `Logging` методу расширения `AddConsole`:

```

var loggingConfig = Configuration.GetSection("Logging");
loggerFactory.AddConsole(loggingConfig);
loggerFactory.AddDebug();

```

Обратите внимание: здесь выполняются два вызова, и на первый взгляд может быть непонятно, что происходит. Прежде всего стоит заметить, что некоторые методы расширения обладают большими возможностями, чем другие. Два таких метода используются для активизации журналирования. Первый добавляет вывод журнала на консоль, а второй — в окно вывода `Debug` в Visual Studio. Вызов `AddConsole` позволяет передать экземпляр `IConfiguration`, что мы делаем в приведенном блоке кода. С другой стороны, методу `AddDebug` этот экземпляр не передается. Это означает, что `AddConsole` имеет возможность использовать настройки, сохраненные для фильтрации сообщений по пространствам имен. `AddDebug` не имеет перегруженной версии, позволяющей передать настройки фильтрации, поэтому команда Alpine решила написать собственную реализацию.

```

loggerFactory.AddDebug((className, logLevel) => {
    if (className.StartsWith("AlpineSkiHouse."))
        return true;
    return false;
});

```


Фильтру требуется делегат `Func`, получающий строку с категорией (обычно полностью уточненное имя класса) и уровнем `LogLevel` сообщения. Просмотреть само сообщение не удастся, да это и не нужно, так как это вы всего лишь определяете, нужно ли сохранять в журнале сообщения из этой категории в данный момент времени. В этой тривиальной реализации фильтра журнала команда решила выводить сообщения в окно `Debug` только в том случае, если они поступают из пространства имен, которое создали они сами.

Применение областей действия

Области действия (`scopes`) помогают лучше понять, откуда взялась информация в журнале, позволяя раскрыть часть цепочки вызовов. Область действия создается автоматически для каждого запроса, причем разные части конвейера исполнения могут создавать собственные вложенные области действия. В процессе переходов по запросу новые области действия открываются и закрываются, словно матрешки. Вы также можете добавить собственную область действия и обернуть в нее процесс, охватывающий несколько классов. Область действия можно открыть перед входом в критический аспект бизнес-процесса: выполнение всей работы, необходимой для завершения покупки и получения оплаты в стороннем сервисе. Такая область действия закрывается при завершении операций. В ASP.NET Core срок жизни областей действия ограничивается сроком жизни запроса HTTP.

Команда Alpine захотела более четко понять действия, выполняемые для каждого входящего запроса, особенно когда этот запрос связан с добавлением данных новой карты. Для этого разработчики сначала изменили настройки для включения информации области действия в `appsettings.json`, задав параметру значение `"IncludeScopes": true`. В результате в журнале появляется информация области действия:

```
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[1]
      => RequestId:0HKV5MFUPFPVU RequestPath:/
      Request starting HTTP/1.1 GET http://localhost:5000/
```

Информации области действия предшествует один или несколько знаков `=>`, а единственной областью действия, активной в этой точке, является корень запроса. Вложенная форма встречается при присоединении к выводу дополнительных сегментов. Это становится более очевидным для записей в журнале, появившихся из-за действий с контроллером:

```
info: Microsoft.AspNetCore.Authorization.DefaultAuthorizationService[1]
      => RequestId:0HKV5MFUPFPVU RequestPath:/skicard
      => AlpineSkiHouse.Web.Controllers.SkiCardController.Index (AlpineSkiHouse.Web)
      Authorization was successful for user: james@jameschambers.com.
```

Для дальнейшего прояснения информации области действия команда решила создавать собственные области действия для ключевых бизнес-сценариев. Ниже

приведен пример из класса `SkiCardController`, где в нестандартную область действия заключается код, предназначенный для добавления карты к учетной записи пользователя. Обратите внимание на вызовы `_logger` и на то, где они расположены относительно начала области действия `CreateSkiCard`:

```
if (ModelState.IsValid)
{
    var userId = _userManager.GetUserId(User);
    _logger.LogDebug($"Creating ski card for {userId}");
    using (_logger.BeginScope($"CreateSkiCard:{userId}"))
    {
        SkiCard skiCard = new SkiCard
        {
            ApplicationUserId = userId,
            CreatedOn = DateTime.UtcNow,
            CardHolderFirstName = viewModel.CardHolderFirstName,
            CardHolderLastName = viewModel.CardHolderLastName,
            CardHolderBirthDate = viewModel.CardHolderBirthDate.Value.Date,
            CardHolderPhoneNumber = viewModel.CardHolderPhoneNumber
        };

        _skiCardContext.SkiCards.Add(skiCard);
        await _skiCardContext.SaveChangesAsync();
        _logger.LogInformation($"Ski card created for {userId}");
    }
    _logger.LogDebug($"Ski card for {userId} created successfully, redirecting to
Index...");
    return RedirectToAction(nameof(Index));
}
```

Вывод в журнал при создании карты пользователем выглядит примерно так (с явным указанием того, к какой точке цепочки вызовов относятся сообщения):

```
dbug: AlpineSkiHouse.Web.Controllers.SkiCardController[0]
=> RequestId:0HKV5MFUPFQ01 RequestPath:/SkiCard/Create
=> AlpineSkiHouse.Web.Controllers.SkiCardController.Create (AlpineSkiHouse.Web)
    Creating ski card for f81d8803-304a-481b-ad4f-13ef7bcec240
info: Microsoft.EntityFrameworkCore.Storage.Internal.
    RelationalCommandBuilderFactory[1]
=> RequestId:0HKV5MFUPFQ01 RequestPath:/SkiCard/Create
=> AlpineSkiHouse.Web.Controllers.SkiCardController.Create (AlpineSkiHouse.Web)
=> CreateSkiCard:f81d8803-304a-481b-ad4f-13ef7bcec240
    Executed DbCommand (1ms)...
    ...WHERE @@ROWCOUNT = 1 AND [Id] = scope_identity();
info: AlpineSkiHouse.Web.Controllers.SkiCardController[0]
=> RequestId:0HKV5MFUPFQ01 RequestPath:/SkiCard/Create
=> AlpineSkiHouse.Web.Controllers.SkiCardController.Create (AlpineSkiHouse.Web)
=> CreateSkiCard:f81d8803-304a-481b-ad4f-13ef7bcec240
    Ski card created for f81d8803-304a-481b-ad4f-13ef7bcec240
dbug: AlpineSkiHouse.Web.Controllers.SkiCardController[0]
```

```
=> RequestId:0HKV5MFUPFQ01 RequestPath:/SkiCard/Create  
=> AlpineSkiHouse.Web.Controllers.SkiCardController.Create (AlpineSkiHouse.Web)  
Ski card for f81d8803-304a-481b-ad4f-13ef7bcec240 created successfully,  
    redirecting to Index...
```

Области действия предоставляют информацию о вызовах SQL через Entity Framework, о контроллере и действии, выполняемом как часть запроса, и о идентификаторе запроса. Вся эта информация чрезвычайно полезна для понимания того, что произошло в ходе обработки запроса, а ее ценность дополняет тот факт, что все обращения к базе данных, производимые в контексте действия `SkiCardController.Create`, проще идентифицировать. При разумном использовании это поможет понять суть происходящего.

И последнее замечание по поводу областей действия: используйте их осматривательно и только в том случае, когда они дают вам то, что не может быть получено другими способами. Если идентификатор запроса — все, что вам необходимо для изоляции серии взаимосвязанных событий в журнале, не увлекайтесь созданием стеков областей действия и не усложняйте запись в журнал, если без этого можно обойтись.

Структурированное журналирование

Концепция структурированного журналирования позволяет хранить данные, записываемые в журнал, отдельно от самого сообщения в журнале. Она вводит понятия журнального шаблона и параметров, но при этом поддерживает традиционные текстовые записи в журнале посредством наполнения шаблонов данными. В результате структурирования в журнале создаются записи в формате, удобном для машинной обработки, и это открывает действительно интересные возможности.

Структурированное журналирование использует «постоянный, предопределенный формат сообщений, содержащий семантическую информацию [...]» для извлечения более глубокой аналитики из журнальных данных.

ThoughtWorks Technology Radar, январь 2014 г.

Команда Alpine выбрала библиотеку Serilog и добавила нужные пакеты, а именно пакеты `Serilog.Extensions.Logging` и `Serilog.Sinks.Literate`. Вы найдете их в файле `project.json` проекта, в котором также содержатся другие журнальные зависимости. Пакеты расширений — то, что позволяет Serilog применить свои структурированные возможности с журнальными абстракциями ASP.NET Core, тогда как библиотека `Literate` предоставляет намного более ясный, удобочитаемый консольный вывод.

Следующий шаг — обновление класса `Startup` для использования нового пакета: из него удаляется конфигурация `LoggerFactory`, включая вызов `AddDebug`, которая заменяется следующим фрагментом:

```
Log.Logger = new LoggerConfiguration()
    .MinimumLevel.Information()
    .MinimumLevel.Override("AlpineSkiHouse", Serilog.Events.LogEventLevel.Debug)
    .Enrich.FromLogContext()
    .WriteTo.LiterateConsole()
    .CreateLogger();
```

Эта конфигурация предоставляет те же уровни сообщений об ошибках, которые были добавлены ранее для консольного вывода, и открывает доступ к Serilog приложению во время выполнения.

Использование Serilog с Configuration

Если вы хотите, чтобы ваша конфигурация Serilog отличалась в зависимости от среды, используйте пакет `Serilog.Settings.Configuration`, который позволяет прочитать значения из собственной структуры Serilog. Конфигурация, описанная выше, представляется в `appsettings.json` следующим образом:

```
"Serilog": {
  "Using": ["Serilog.Sinks.Literate"],
  "MinimumLevel": {
    "Default": "Information",
    "Override": {
      "AlpineSkiHouse": "Debug"
    }
  },
  "WriteTo": [
    { "Name": "LiterateConsole" }
  ],
  "Enrich": ["FromLogContext"],
  "Properties": {
    "Application": "Sample"
  }
}
```

С заранее настроенной реализацией `IConfigurationRoot` достаточно включить в метод `Configure` класса `Startup` следующий код для загрузки настроек журнала:

```
var logger = new LoggerConfiguration()
    .ReadFrom.Configuration(Configuration)
    .CreateLogger();
```

У вас появляется возможность использовать то, что вы узнали ранее о конфигурации на уровне отдельных сред, и применить новые знания в своих настройках записи в журнал, переопределяя настройки в скрытых данных пользователей, переменных среды или в файлах `appsettings.EnvironmentName.json` по мере необходимости.

В качестве простого примера того, как группа может подходить к записи в журнал, рассмотрим команду записи в журнал в методе `Contact` контроллера `Home`. В команду включается журнальный шаблон и выводимые данные:

```
_logger.LogDebug("User visited contact page at {VisitTime}", DateTime.Now);
```

Место, в которое ведется запись, может рассматриваться как «приемник». Вы можете завести сколько угодно приемников и вести запись либо во все сразу, либо по отдельности — в те приемники, которые вы выберете. Когда запись ведется в неструктурированные приемники (например, при выводе в консольное окно или в текстовый файл), приведенный вызов `LogDebug` просто воспроизводится в виде текста. Но при выводе в приемник, который «понимает» параметризованную версию записи, преимущества структурированной записи начинают проявляться. Рассмотрим один из таких приемников, а заодно взглянем на журналирование под другим углом.

Журналирование как сервис

Чтобы более эффективно использовать возможность интерпретации содержимого журнала как данных, необходимо иметь приемник, который «понимает» сохраняемую информацию. Что еще важнее, он помогает затем извлечь эту информацию.

Когда вы закладываете возможность использования многих сервисов в вашем приложении, сбор информации начинает играть более важную роль. Кроме того, когда эти сервисы распределяются по разным серверам, собирать нужную информацию становится труднее. Для борьбы со сложностями и преодоления рисков, связанных с задержками, для сбора журнальных данных чаще всего используется отдельный сервис.

Команда `Alpine` решила работать с `Seq`; этот сервер бесплатен для локального использования разработчиками, но требует коммерческой лицензии для использования при эксплуатации приложения. И хотя мы не будем слишком глубоко заходить в описание конкретных продуктов или сервисов, в этом случае выбрано именно ядро `Seq`, потому что его написали те же люди, что и `Serilog`, оно легко подключается и взаимодействует с `Serilog`.

За новейшей информацией о загрузке и установке `Seq` обращайтесь по адресу <https://getseq.net/>.

Так как библиотека `Serilog` уже была настроена, команде `Alpine` оставалось только добавить дополнительный пакет `Serilog.Sinks.Seq`, а затем добавить дополнительный приемник в конфигурацию записи в журнал в `Startup`:

```
Log.Logger = new LoggerConfiguration()  
    .MinimumLevel.Information()  
    .MinimumLevel.Override("AlpineSkiHouse", Serilog.Events.LogEventLevel.Debug)
```

```
.Enrich.FromLogContext()
.WriteTo.LiterateConsole()
.WriteTo.Seq("http://localhost:5341")
.CreateLogger();
```

В этой конфигурации следует обратить внимание на вызов `WriteTo.Seq()`, при котором местонахождение сервера Seq, используемого для записи, передается в параметре; `localhost:5341` — значение по умолчанию для локальной установки Seq. При следующем запуске приложения информация начинает появляться в интерфейсе Seq (рис. 12.3).



Рис. 12.3. Веб-интерфейс Seq

Информация в журнале стала более наглядной. Важные сведения выделены, а разные уровни журналирования обозначаются разными цветовыми индикаторами. Пользовательский интерфейс может предоставить визуальные признаки, но самое замечательное — это возможность углубленного анализа отдельных записей. На рис. 12.4 показано, как выглядит развернутое сообщение `LogDebug` для действия `Contact`.

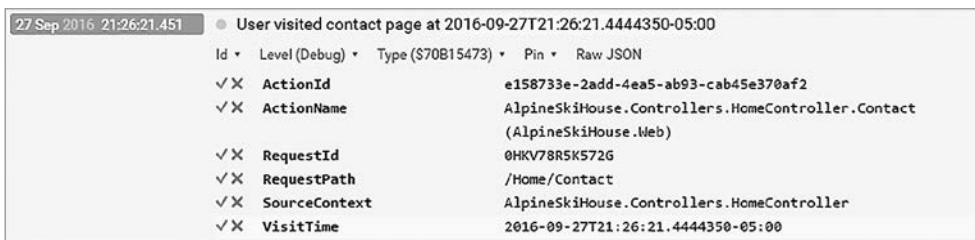


Рис. 12.4. Развернутая запись в журнале

Видите параметр `VisitTime`, переданный при формировании записи? Мы добились желаемого — интерпретировали содержимое журнала как данные; доказательством служит возможность проведения поиска в данных, предоставляемая инструментарием. На рис. 12.5 приведен пример поиска с запросом, в котором в качестве критерия поиска используется поле `RequestId`.

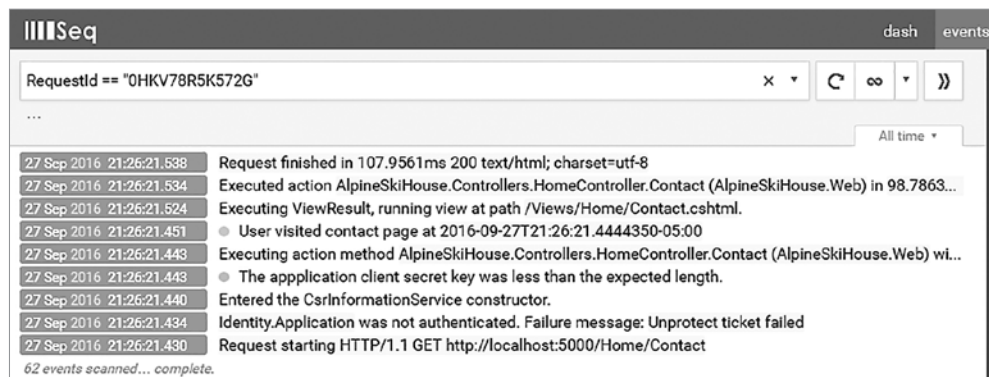


Рис. 12.5. Поиск по идентификатору запроса

Благодаря структурированной информации и мощному журнальному сервису мы смогли легко отобразить все записи в журнале, соответствующие заданному набору поисковых критериев. Представьте, какую пользу принесет эта возможность при поиске информации, связанной с использованием определенного ресурса или переходом пользователя между несколькими экранами. Также представьте, насколько упрощается ваша задача, если вам не придется искать нужную информацию в нескольких файлах журналов на разных серверах приложений!

Итоги

Группы разработки работают во все более тесном контакте с отделами эксплуатации, и в мире, в котором среды создаются и уничтожаются «на лету», таким традиционным аспектам работы приложений, как конфигурация и журналы, тоже приходится подстраиваться под это. В ASP.NET Core разработчик лучше, чем когда-либо, подготовлен к реализации структур конфигурации и журналирования в своих приложениях. Эти структуры могут формироваться под воздействием входных данных и привязываться к конкретным контекстам. Объединение всех составляющих и использование журнального сервиса поможет быстрее получить необходимую информацию для диагностики возникающих проблем, а современные библиотеки и сервисы повысят эффективность работы наших команд. В следующей главе более подробно рассматриваются вопросы идентификации, безопасности и управления правами в ASP.NET.

ЧАСТЬ III

Спринт третий: В пасти у зверя

Глава 13. Идентификация, безопасность и управление правами.....	246
Глава 14. Внедрение зависимостей.....	278
Глава 15. Роль JavaScript	289
Глава 16. Управление зависимостями	313
Глава 17. Стильный интерфейс.....	327
Глава 18. Кэширование	348

На 3-м сезоне «Секретных материалов» Даниэль спала немного лучше, чем на предыдущем. Никаких гибридов человека с червем больше не было, и это вселяло надежды. Гибридов человека с инопланетными пришельцами было предостаточно, но Даниэль они не беспокоили. Похоже, единственное, что их интересовало — устанавливать крошечные наблюдательные устройства в носу у людей. Если инопланетянам хотелось знать, когда она сморкается, — ее это устраивало. Последний спринт был очень продуктивным; они разобрались с безопасностью обращений к данным, журналом и внедрением зависимостей, и у них был сайт, готовый к развертыванию.

«Всем привет, — сказал Балаш. — Переходим к ретроспективе номер 2. Но прежде чем приступить, давайте оглянемся и посмотрим, удалось ли нам решить некоторые проблемы во время спринта 1. — Он вытащил блокнот и перелистал страницы: «Такое чувство, что продукт мы еще не выдали. Как дела на этом фронте?»

«Да вроде неплохо, — сказала Кэндис. — У нас уже готово несколько страниц. Выглядят пока не очень, но зато мы проработали структуру».

«Работа с базой данных тоже в норме, — сказал Марк-2. — С Entity Framework выборка и добавление данных выполняются проще простого. Я даже начинаю понимать, как работают асинхронные методы».

«А мне нравится поддержка запросов LINQ, — добавила Даниэль. — Она уже была в NHibernate, но всегда смотрелась немного странно. EF выглядит так, словно при проектировании ориентировались на LINQ, а в NHibernate поддержка была словно прилеплена сверху».

«Я просто счастлив, — вступил в разговор Тим, которому вообще не полагалось участвовать в собраниях группы. — Работа над продуктом еще не закончена, но сборка осуществляется по десятку раз в день, и каждая новая версия — шаг вперед по сравнению с предыдущей. Большей частью».

«Смотрите, — сказал Адриан, — по-моему, розово-красная цветовая схема отлично смотрится. Напоминает пирожные с кремом».

Команда рассмеялась — бедного Адриана уже неделю ругали за изменение стилового оформления интерфейса. «Хорошо, — продолжал Балаш, — с одной проблемой справились. Также у нас проходят обеденные семинары. На этой неделе их было два, спасибо Даниэль и Марку за организацию».

«Да уж, спасибо, — сказал Тим, похлопывая себя по животу. — Я мало что понял, но мне показалось, что мне стоит присутствовать... из солидарности».

«А теперь к доске — посмотрим, какие предложения у нас будут на этой неделе». Балаш разочарованно посмотрел на новые маркеры, которые ничем не пахли. После предыдущей встречи кто-то пожаловался, и действительно хорошие маркеры убрали. Ходили слухи, что их закопали где-то в окрестностях.

Участники снова взяли маркеры и начали писать на доске.

Начать	Прекратить	Продолжать
Разобраться с JavaScript	Делать уродливые страницы	Использовать .NET
Подготовить больше автоматизированных тестов	Держать регистрационные данные на локальной машине, это сильно замедляет работу	Использовать внедрение зависимостей
Повторно использовать код, встречающийся в нескольких местах		Обеденные семинары
Использовать более мощный контейнер для внедрения зависимостей		
Сделать сайт посимпатичнее		

«На этой неделе не так много, — сказал Балаш. — Похоже, мы неплохо сработались. Меня интересует эта запись про автоматизированные тесты; кто это написал?»

«Это я, — сказал Адриан. — Я видел, как пара проблем возникла снова уже после того, как мы их исправили. Было бы неплохо запустить тесты, чтобы такого больше не было».

«Отличная идея, за нее я даже готов простить этот красно-розовый кошмар. Присоединяюсь! — сказал Марк-2. — Я предпочитаю проводить тестирование во время работы над кодом, но у нас здесь вечный аврал, и мне не удастся выкроить время на написание хороших тестов».

«Ты говоришь о разработке через тестирование?» — спросил Балаш.

«Точно! — ответил Марк-2. — Тесты нужно писать еще до того, как мы пишем код. Такой код лучше адаптируется к изменениям, а заодно дает дополнительную страховку».

«Сначала пишем тесты? — спросила Даниэль. — Не понимаю».

«Хорошо, — прервал Балаш. — Я думаю, это станет хорошей темой для обеденного семинара на этой неделе. С внедрением зависимостей у нас тоже не все благополучно?»

«Похоже, что так, — сказал Честер. — Встроенный контейнер неплох, но с некоторыми вещами он справляется не так хорошо, как что-нибудь типа Autofac».

«Хмм, я думаю, мы добавим это в спринт. Возьмешь это направление, Честер?» — спросил Балаш.

«Хорошо», — ответил Честер.

«Порядок, — продолжал Балаш. — Посмотрим, можно ли что-нибудь сделать с оформлением сайта. Я знаю — странно беспокоиться об этом, когда готова еще не вся функциональность, но люди лучше относятся к сайту, который хорошо выглядит. Кстати, руководство тоже. Все сначала смотрят на внешний вид, так дадим им что-нибудь такое, на что смотреть приятно».

«Наконец, нам нужно определиться с подходом к JavaScript. Я видел пару правок, связанных с JavaScript, и они выглядели сыровато. Возможно, это можно делать иначе, более цивилизованно», — предположил Балаш.

«Я согласен, — сказал Честер. — У меня такое чувство, что у нас возникнут проблемы с сопровождением имеющегося кода, когда сайт станет больше. Мы работаем с JavaScript так, словно сейчас 2002 год — одиночные функции, встроенные в сценарные теги на странице».

«Здорово, — сказал Балаш. — Займись этим, Честер. А я поработаю над картами для следующего спринта. Давайте расходиться по домам, время уже позднее».

Даниэль не нужно было уговаривать — ее ждал недосмотренный эпизод «Секретных материалов» и салат с киноа, который весь день мариновался в смеси оливкового масла с кориандром. За выходные она собиралась побывать на озере — вероятно, последний раз за сезон, потому что погода начинала меняться.

13

Идентификация, безопасность и управление правами

Вмешиваться было не в ее привычках, но Даниэль ощутила, что за последний месяц возникла пугающая тенденция. Этим летом разработчики один за другим уходили в отпуск, чтобы отдохнуть и подготовиться к официальному запуску новой программной системы. За последние две недели она не раз слышала, как другие участники говорили на темы безопасности, но до сих пор не придавала этому значения.

«Хорошо, это разумно, — сказал Марк-1 в ответ на реплику Марка-2, — потому что я могу просто включить это в качестве контрольной точки промежуточного ПО. Но как сделать, чтобы аутентификация Facebook работала локально?»

«О, это просто, — сказал Марк-2. — Я получил от Тима сертификаты подлинности страницы нашего курорта на Facebook и сохранил их в своей ветви. Загрузи их локально, все будет работать».

«Одну минуту» — прервала их Даниэль. — Ты хочешь сказать, что сертификаты, которые будут использоваться в рабочей версии, хранятся на GitHub?»

«Да, но это неважно. У нас закрытый репозиторий», — ответил Марк-2.

«Это понятно, — возразила Даниэль. — Но здесь возникают по крайней мере две проблемы. Во-первых, вопрос ответственности: тот факт, что сертификаты рабочей версии хранятся в репозитории, означает, что они доступны для любого участника нашей группы независимо от того, доверяем мы ему или нет. Если он загрузит данные из репозитория на взломанную машину, наши данные могут украсть. Кроме того, я вообще не хочу оказаться в положении, в котором мне нужно знать итоговые ключи доступа для любой части системы!» Оба Марка кивнули, потому что понимали, к чему она клонит.

«Согласен... Я работал в одном месте, где была такая проблема, — добавил Марк-1. — Но как тогда мне добиться, чтобы все это работало на моей машине?»

«Может, пора заняться скрытием пользовательских данных?» — предложил Марк-2.

«Да, и нам, вероятно, стоит собрать группу, чтобы обсудить, как мы будем двигаться в этом направлении. Итак, кто попросит Тима сбросить ключи доступа Facebook?»

Эшелонированная оборона

Вместо того чтобы погружаться в технические аспекты безопасности, в этой главе мы рассмотрим «общую картину». В книге, которая вроде бы предназначена для технических специалистов, это может показаться противоестественным, но не все важные аспекты имеют технологическую природу.

Конечно, технологическая часть очень важна. Прикрыть свой сайт сертификатом SSL и назвать его «безопасным» никогда не было достаточно. В игре участвуют разные элементы на многих уровнях, и ваш профессиональный долг — задействовать системы, протоколы и механизмы, которые предотвращают не только технический взлом, но и нарушения безопасности из-за человеческого фактора. Приложения и данные, накопленные нашей компанией, необходимо защищать — но делать это так, чтобы они могли использоваться руководством, администраторами, сотрудниками компании и другими разработчиками. Думать о том, как использование данных может создать дополнительный риск для компании, не менее важно, чем думать о возможностях использования данных.

Компании тратят миллионы долларов на брандмауэры, шифрование и устройства безопасного доступа. Но все эти деньги тратятся впустую, потому что ни одна из этих мер не направлена на самое слабое звено в цепочке безопасности.

Кевин Митник, свидетельские показания в Комитете Сената по вопросам государственного управления, март 2000 г.

Широта охвата этой книги недостаточна, чтобы помочь вам в защите ваших приложений от хорошо спланированных социальных проникновений или инфраструктурных реализаций; тем не менее мы можем обсудить возможности проектирования приложений, серьезно относящихся к защите предоставляемых данных. Понимая контекст, в котором приложение может стать уязвимым для атаки, мы можем лучше справиться с наиболее очевидными уязвимостями, которые могут стать легкой добычей потенциальных злоумышленников.

Внутренние угрозы

Команда разработчиков из Alpine Ski Resort пребывала в полной уверенности, что их кодовая база находится в безопасности, потому что они доверяли друг другу конфиденциальные данные и полагали, что их исходный код в закрытом репозитории GitHub надежно защищен.

Однако мы часто забываем (особенно в небольших группах), что сама природа нашего доверия часто недолговечна. Состав группы может быстро измениться с приходом практиканта, увольнением опытного работника или изменением

в структуре отдела. Когда в игру вступают такие изменения, вы понятия не имеете, как могут использоваться данные, доступные на сегодняшний день. Следовательно, необходимо позаботиться о том, чтобы ключи рабочей версии (и даже ключи доступа из сред более низкого уровня) не проникали в систему управления исходным кодом. Придерживаясь принципа наименьших привилегий, вы можете позволить работникам выполнять их работу, не предоставляя им доступ к ключам. Несоблюдение этих принципов повышает вероятность раскрытия секретной информации из-за корысти или недовольства работников, или из-за нарушения системы безопасности вследствие заражения машины разработчика вирусом. Если такой работник не имеет доступа к ключам, последствия взлома с большой вероятностью будут изолированы.

Многие сторонние сервисы предоставляют ключи доступа API уровня разработчика и маркеры (tokens) доступа, которые позволяют хранить коды для эксплуатации продукта отдельно от среды разработки. Вообще говоря, раздельное хранение в интересах таких сторон, потому что оно помогает защитить данные и предоставляемые ими услуги. Если вариант с получением ключей разработчика недоступен напрямую, обратитесь за помощью к стороне, предоставляющей услугу! Вы можете защитить своих разработчиков, просто не предоставляя им информацию, которая может стать мишенью атаки. Если рабочие маркеры доступа не хранятся в репозитории, то они не попадут в руки злоумышленника. Маркеры доступа рассматриваются позднее в разделе «Скрытые данные пользователей» этой главы.

Другие направления атаки с эксплуатацией уязвимостей или злонамеренным использованием услуг приложения в основном связаны с избыточными правами доступа, предоставляемыми API. К этой категории могут относиться уязвимости, созданные непреднамеренно (например, если API используется во внутренней работе, считается безопасным и поэтому не требует аутентификации). Аутентификация — полезная мера, но приложение или сервис, предоставляющие проверенному пользователю неограниченный доступ к системе, также создают риск. При наличии правильных инструментов атакующий может использовать сервисы приложения непредвиденным способом (например, в обход веб-интерфейса передать свое имя пользователя и пароль для прямого обращения к API).

При обсуждении идентификации, утверждений и политик в этой главе вы узнаете, как лучше отделить авторизацию от аутентификации для защиты ваших API.

Следует упомянуть еще об одной разновидности рисков: техническом долге. Слишком часто встречаются злоупотребления, невозможные в нормальной ситуации, которые возникают по единственной причине: из-за стремления побыстрее выдать некоторую функциональность без должного анализа безопасности и целостности данных. Концепция некорректной привязки уже рассматривалась в главе 3 «Модели, представления и контроллеры»; там было показано, что простое использование одной модели для чтения и записи создает уязвимость. Честный, непредвзятый анализ программного продукта часто открывает дефекты,

о которых вашей группе разработки уже известно, и для этого даже не нужно погружаться в анализ кода.

Внешние угрозы

Даже когда сервис предназначен для внутреннего использования, вероятно, в какой-то момент он будет потреблять данные из внешних источников или взаимодействовать с другими сервисами API в интересах пользователя или для реализации бизнес-сценариев. К сожалению, мы живем в мире, в котором приходится думать о том, как происходят эти взаимодействия и в какой степени можно доверять обрабатываемым данным, вызывающим сторонам или сервисам, с которыми мы взаимодействуем.

Вы импортируете данные? Важно знать, откуда эти данные берутся. Вы проверяете весь файл или набор записей либо ограничиваетесь проверкой отдельных строк? Некоторые атаки возможны из-за того, что каждая строка по отдельности действительна, а файл или операция, определяемая файлом, является мошеннической. Пользователи могут создавать или изменять документы в вашем приложении? В большинстве таких ситуаций происходят операции с ресурсами, а следовательно, к ним следует применять соответствующую политику приложения.

Вы предоставляете доступ к своим данным партнерским сайтам? Это совершенно законная и часто результативная практика, которая способна выделить вашу организацию на фоне конкурентов. Включаете ли вы при этом проверки, гарантирующие, что сайту, использующему ваш API, можно доверять? Все основные браузеры поддерживают возможность блокировки сценариев из загружаемых данных, если серверу неизвестно, откуда поступил вызов.

Скрытые данные пользователей

Вернемся к команде разработчиков из Alpine Ski Resort. Если кто-то в руководстве раздает ключи доступа команде разработчиков, вы должны как минимум насторожиться. Выглядит все так, словно всем кассирам (и возможно, ночному охраннику) раздаются коды от банковского хранилища в подвале. Если руководство разбирается в бизнесе, оно относится к рабочим ключам доступа с гораздо большим вниманием.

В основном скрытые данные пользователей появляются так: у разработчиков существует необходимость в надежной разработке и локальном использовании приложения, обладающего доступом к защищенным ресурсам и сервисам. Вместо того чтобы принимать ответственность за конфиденциальные ключи доступа, данные и сервисы, разработчики регистрируют собственный экземпляр сервиса приложения (например, Facebook) и локально используют собственные ключи. Для этого разработчикам нужен механизм хранения сертификатов, их загрузки

в приложении при выполнении и предотвращения их совместного использования с другими разработчиками. Это особенно справедливо, когда использование ключей неуполномоченными сторонами создает издержки или вызывает другие побочные эффекты; поэтому разработчики должны прикладывать усилия для охраны своих маркеров и ключей.

Другое преимущество, которое должно побудить разработчиков использовать приватные ключи, привязанные к их учетным записям, относится к использованию ресурсов, контролируемости и затратам. И хотя в высокоуровневых средах обычно поощряется использование «песочниц» (изолированных сред), предоставляемых разработчиками сервисов, вы должны проанализировать, существуют ли ограничения на взаимодействие разработчиков с API. Если такие ограничения существуют, то при обращении к таким сервисам с большой вероятностью следует использовать уникальные ключи для каждого разработчика.

Скрытые данные пользователей (user secrets) — что-то вроде переменных среды для отдельных пользователей на уровне отдельных проектов. Существуют значения конфигурации, изолированные от проекта, но доступные во время выполнения через систему конфигурации ASP.NET Core. Скрытые данные пользователей хранятся в профиле пользователя в файле JSON и чаще всего содержат пары «ключ — значение», хотя, как вы узнали в главе 12 «Конфигурация и журналирование», возможны и более сложные структуры.

Основная операция по добавлению скрытых данных выполняется достаточно тривиально:

```
dotnet user-secret set YourKey "нужное значение"
```

На момент написания книги набор команд для взаимодействия со скрытыми данными пользователей в интерфейсе командной строки CLI был ограничен:

- **set**: заданное значение связывается с заданным ключом.
- **list**: вывод списка всех известных секретных данных пользователя в проекте.
- **remove**: удаление заданной пары «ключ — значение» из хранилища.
- **clear**: удаление всех ключей из хранилища без подтверждения.

Команда `user-secret` может выполняться только в каталоге, в котором существует файл `project.json`, и файл должен содержать значение для ключа `userSecretsId`; в противном случае произойдет ошибка выполнения. Команда использует значение `userSecretsId` для обращения к конкретной папке на диске, находящейся в вашем профиле пользователя.

Скрытые данные пользователей хранятся в файлах JSON, а следовательно, поддерживают многоуровневую структуру и назначение свойств. Базовые команды, вводимые в CLI, не позволяют легко создавать более сложные скрытые данные с использованием секций. Не существует команд для создания сложных объектов с вложенными свойствами и выполнения таких операций, как удаление несколь-

ких ключей; они быстро становятся трудоемкими. Впрочем, использовать CLI не обязательно, потому что с файлами можно работать в текстовой форме. Этот файл хранится в каталоге `appdata\Roaming\Microsoft\UserSecrets` внутри каталога, имя которого совпадает с вашим значением `userSecretsId`. Также к файлу можно обратиться прямо в Visual Studio; щелкните правой кнопкой мыши на узле проекта в Solution Explorer и выберите команду `Manage User Secrets`.

Аутентификация с поддержкой Azure

В корпоративных средах, в которых пользователи находятся под управлением администраторов и привязываются к внутренним ролям и политикам, на заднем плане часто работает AD (Active Directory). Технология Active Directory проложила путь в облако и стала доступной в виде сервиса, предоставляемого Microsoft Azure. Существуют инструменты для связывания внутреннего экземпляра Active Directory с экземпляром, размещенным в облаке, поэтому вы можете импортировать или перенести свой внутренний каталог в долгосрочное управляемое решение в Microsoft Azure, и даже подключить пользователей из подписки Office 365 вашей организации. Так вы пользуетесь преимуществами межрегиональных средств надежности и преодоления отказов, улучшенным временем безотказной работы, а также более уверенно чувствуете себя, зная, что обновление будет происходить «на лету», а для размещения будет использована хорошо защищенная инфраструктура. Кроме того, появляется возможность добавления таких возможностей, как мобильная поддержка, многофакторная аутентификация и обеспечение удаленной политики для несметного числа устройств.

Варианты конфигурации в Azure Active Directory

При выборе Microsoft Azure Active Directory в качестве механизма аутентификации для вашего приложения вам на выбор предоставляются две конфигурации:

- *Одноклиентская*: приложение подключается к корпоративному каталогу. Пользователи вашего домена могут выполнять вход в соответствии с политикой, установленной администраторами.
- *Мультиклиентская*: приложение настраивается так, чтобы аутентификация в приложении могла обеспечиваться несколькими каталогами вашей учетной записи Microsoft Azure или внешними по отношению к ней. Каждая организация должна предоставить все необходимые данные вашему приложению, но это позволит каждой из этих организаций обращаться к нему.

Microsoft Azure предоставляет информацию о том, как происходит аутентификация, так что в мультиклиентских сценариях у вас есть возможность выбрать для пользователя тот вариант, который уместен в вашем контексте.

Хотя эти возможности могут способствовать успеху внутреннего приложения, в большинстве приложений, создаваемых для внешнего потребителя, должна под-

держиваться самостоятельная регистрация и администрирование. Microsoft Azure Active Directory ориентируется на внутренних пользователей — пользователей, которые выигрывают от централизованного администрирования и которые уже могут иметь учетную запись в организации. Все это не очень хорошо совмещается с типом приложения, которое мы строим для Alpine — приложения, ориентированного на внешнего пользователя.

Пользователи за пределами организации

Когда вы хотите обеспечить персонализацию пользовательских взаимодействий, пользователь должен пройти аутентификацию, но при этом не нужно создавать лишние препятствия и требовать, чтобы учетные записи пользователей создавались администраторами. Также существует другая группа проблем, которые плохо укладываются в одноклиентские или мультиклиентские варианты аутентификации, в контексте рабочей нагрузки или затрат:

- Пользователи должны самостоятельно создавать свои учетные записи и управлять ими, не требуя административного вмешательства для выполнения входа или изменения настроек учетной записи.
- Пользователи могут быть кратковременными посетителями; такие пользователи практически не приносят прибыли организации, быстро покидают сайт и не пользуются его возможностями.
- Пользователи могут захотеть воспользоваться уже имеющимися идентификационными данными в виде учетной записи стороннего ресурса (например, Google или Facebook).
- Пользователи часто используют один и тот же набор сертификатов на разных сайтах. Если защита этих сертификатов будет нарушена в другом, менее надежном месте, их данные и взаимодействия с нашим приложением также могут оказаться под угрозой.

По этим причинам Microsoft Azure также предлагает вариант, называемый Azure Active Directory B2C, который характеризуется такой же высокой доступностью, как и Azure Active Directory, потому что он построен на базе Azure Active Directory, но при этом обладает дополнительным преимуществом — предоставлением источников аутентификации в виде учетных записей социальных сетей. Существует бесплатный тариф для начала работы, позволяющий использовать до 50 000 уникальных пользователей и 50 000 аутентификаций в месяц.

После аутентификации Azure AD B2C перенаправляет вас обратно к приложению с маркером JWT, который содержит информацию о пользователе, полученную из локального хранилища данных или от сервиса социальных сетей с учетом утверждений.

Чтобы начать работу с Azure AD B2C, сначала необходимо создать каталог B2C в подписке Microsoft Azure. При создании каталога обязательно установите флажок B2C (рис. 13.1), потому что после создания изменить его уже не удастся. Возможно, для получения полного доступа к каталогу потребуются несколько минут.

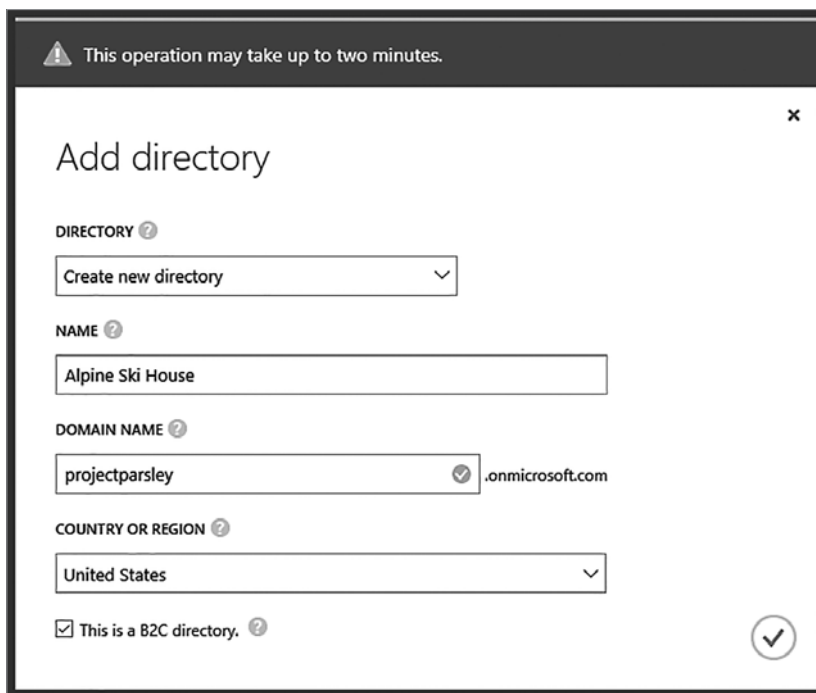


Рис. 13.1. Создание каталога B2C

После того как каталог будет создан, необходимо заполнить ряд параметров конфигурации для полноценной его работы в приложении, а именно добавить провайдеров социальной идентификации, выбрать атрибуты пользователей и добавить необходимые настройки в приложение.

Провайдеры социальной идентификации — такие, как Microsoft Account или Facebook, — требуют, чтобы вы сначала зарегистрировали свое приложение на этих сервисах. За списком мест регистрации для каждого поддерживаемого провайдера обращайтесь по адресу <http://aspnetmonsters.com/2016/09/heres-a-hand-y-list-of-social-login-providers/>. При настройке провайдеров идентификации в Azure AD B2C вам будет предложено заполнить форму, показанную на рис. 13.2, которая дает возможность сервисам взаимодействовать и установить личность пользователя.

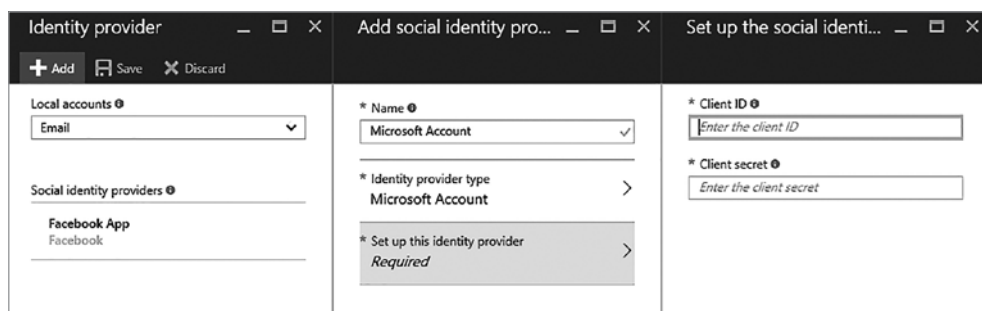


Рис. 13.2. Добавление провайдера социальной идентификации в Azure AD B2C

При регистрации в этих сервисах вы получите идентификатор клиента и секретные данные, которые должны использоваться при настройке сервиса B2C. Интерфейсы разных провайдеров различаются, и поиск необходимых настроек остается вам для самостоятельной работы, но на рис. 13.3 представлен такой интерфейс для средств разработчика приложений Facebook. В данном случае идентификатор клиента называется «идентификатором приложения» (App ID), а клиентские секретные данные называются просто «секретные данные приложения» (App Secret); тем не менее эти названия зависят от сервиса.

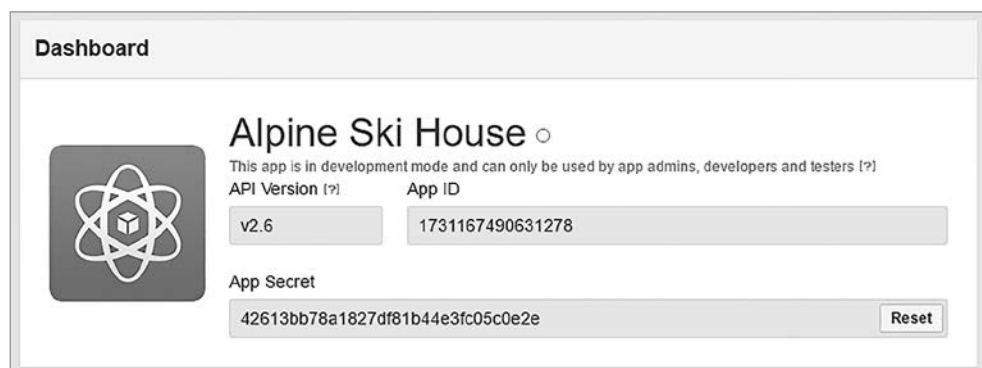


Рис. 13.3. Отображение идентификатора клиента и секретных данных в приложении Facebook

После добавления социальных провайдеров вы можете добавить необязательные атрибуты, специфические для конкретного пользователя, если это уместно для процесса регистрации в вашем приложении. Например, с пользователем можно связать такую пользовательскую настройку, как размер лыж, которые он обычно берет напрокат; тем не менее ввод этой информации не должен быть обязательной частью процесса регистрации. С другой стороны, такие аспекты, как отображаемое имя и адрес электронной почты, предоставляются многими социальными сер-

висами, и эту информацию стоит запросить у пользователя при создании учетной записи. Чтобы сделать эти поля обязательными, необходимо создать политику наподобие показанной на рис. 13.4; нужные атрибуты связываются с регистрационной политикой каталога.

NAME	DATA TYPE	DESCRIPTION	ATTRIBUTE TYPE
City	String	The city in which the user is located.	Built-in
Country/Region	String	The country/region in which the user is located.	Built-in
<input checked="" type="checkbox"/> Display Name	String	Display Name of the User	Built-in
<input checked="" type="checkbox"/> Email Address	String		Built-in
Given Name	String	The user's given name (also known as first name).	Built-in
Job Title	String	The user's job title.	Built-in
Postal Code	String	The postal code of the user's address.	Built-in
State/Province	String	The state or province in user's address.	Built-in
Street Address	String	The street address where the user is located	Built-in
Surname	String	The user's surname (also known as family name or last name).	Built-in

Рис. 13.4. Настройка регистрационной политики по умолчанию для каталога

Если вы хотите использовать в своем приложении атрибуты отображаемого имени (Display Name) и адреса электронной почты (Email Address), вам также придется выполнить настройки в разделе **Application Claims**, чтобы эти значения передавались вашему приложению в маркере JWT. Помимо процесса регистрации, вы также можете создать политики для управления многофакторной аутентификацией, редактирования профиля или сброса пароля. Политики предоставляют мощный механизм для описания средств управления идентификацией, которые вам хотелось бы предоставить своим пользователям.

Наконец, следует сгенерировать значения, необходимые для использования Azure AD B2C в приложениях. Для этого необходимо добавить приложение в конфигурацию клиента, как показано на рис. 13.5. Обратите внимание: также требуется установить флажки **Web App/Web API** и **Implicit Flow**, необходимые для работы механизма OpenID Connect в вашем приложении.

Затем настройки добавляются в приложение. В файле `appsettings.json` добавьте следующие значения в корень конфигурации, как показано в листинге 13.1.

Листинг 13.1. Элементы конфигурации, необходимые для настройки промежуточного ПО Azure AD B2C

```
"AzureAd": {
  "ClientId": "CLIENT_ID_FROM_PORTAL",
  "Tenant": "YOUR_TENANT.onmicrosoft.com",
  "AadInstance": "https://login.microsoftonline.com/{0}/v2.0/.well-known/openid-
```

```
configuration?p={1}",
  "PostLogoutRedirectUri": "https://localhost:44392/",
  "SignUpPolicyId": "B2C_1_YOUR_SIGN_UP_POLICY_NAME",
  "SignInPolicyId": "B2C_1_YOUR_SIGN_IN_POLICY_NAME"
}
```

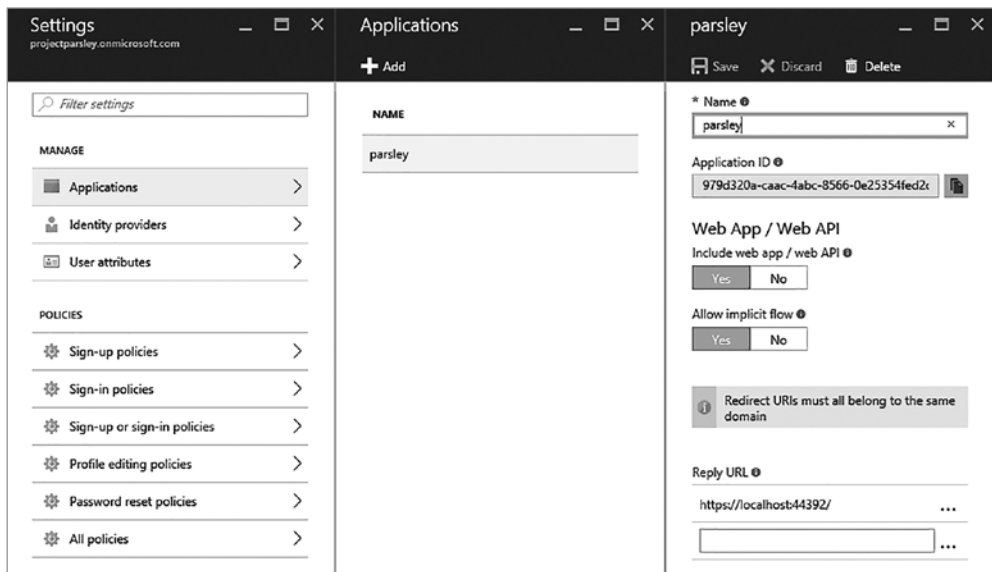


Рис. 13.5. Регистрация приложения в Azure AD B2C

Замените условные значения вашими собственными и не забудьте назначить подходящую конечную точку для URI перенаправления. Как упоминалось ранее в этой главе, если вы работаете с конфиденциальными значениями, подумайте об использовании скрытых данных пользователей. С этими настройками дальнейшим шагом будет добавление пакетов для поддержки OpenID Connect в файле `project.json`. Включите следующие пакеты:

```
Microsoft.AspNetCore.Authentication.Cookies
Microsoft.AspNetCore.Authentication.OpenIdConnect
```

Теперь у вас есть все необходимое для настройки промежуточного ПО аутентификации для вашего сайта, а именно скрытые пользовательские данные, загружаемые при запуске, и пакеты аутентификации для обращения к шлюзу OpenID Connect. Это необходимо сделать для каждой политики, создаваемой в Azure AD B2C. Чтобы поддерживать две политики, добавленные в конфигурации листинга 13.1, необходимо создать два вызова `UseOpenIdConnectAuthentication` с использованием данных, добавленных в конфигурацию, и настроить промежуточное ПО для каждой политики. Это делается примерно так:

```
app.UseOpenIdConnectAuthentication(signUpPolicy));  
app.UseOpenIdConnectAuthentication(signInPolicy));
```

Наконец, необходимо создать или изменить `AccountController` для поддержки необходимых функций идентификации. Сюда входят операции регистрации, входа и выхода. Например, для обработки регистрации можно включить в метод `SignUp` следующий код:

```
var policyId = Startup.SignUpPolicyId.ToLower();  
var authProperties = new AuthenticationProperties { RedirectUri = "/" };  
await HttpContext.Authentication.ChallengeAsync(policyId, authProperties);
```

Этот код сохраняет значение `policyId` из класса `Startup`, создает экземпляр объекта `AuthenticationProperties`, после чего отвечает на запрос вызовом HTTP. Аналогичным образом реализуются действия для других политик.

ПРИМЕЧАНИЕ

Область конфигурации и реализации на момент написания книги еще продолжала развиваться, и хотя механики остаются неизменными, сам код будет изменяться. Возможно, по мере становления B2C появится промежуточное ПО и поддержка упрощенной конфигурации для предложения в целом. За последней информацией об интеграции Azure AD B2C в приложения и полноценным примером кода для работы с ASP.NET Core MVC обращайтесь по адресу <https://azure.microsoft.com/en-us/documentation/services/active-directory-b2c/>.

Каждая организация и, возможно, каждый проект в организации должны принимать решения о том, как провести идентификацию пользователя. Azure Active Directory и Azure AD B2C предоставляют ряд вариантов, не отягощая вас необходимостью создавать хранилище идентификационных данных в приложении и управлять им. В Alpine Ski House ожидается приблизительно несколько сотен пользовательских входов за день. Наши политики, роли и ограничения доступа достаточно статичны, и мы не верим, что добавление внешней зависимости принесет пользу для проекта. Вместо этого мы решили использовать компоненты, встроенные в шаблон проекта, наряду с шаблонами, доступными через NuGet.

Идентификация в ASP.NET Core MVC

ASP.NET Core предоставляет средства аутентификации, авторизации и управления пользователями через встроенный механизм ASP.NET Core Identity. Хотя Identity предоставляет огромное количество настроек конфигурации, большинству приложений достаточно минимальных изменений в настройках по умолчанию.

Основные сущности модели ASP.NET Core Identity — `IdentityUser`, `IdentityRole`, `IdentityUserClaim`, `IdentityRoleClaim`, `IdentityUserLogin` и `IdentityUserToken`. При настройке с использованием Entity Framework, как в веб-приложении Alpine

Ski House, эти сущности хранятся в таблицах (рис. 13.6). Все таблицы ASP.NET Core Identity снабжаются префиксом `AspNet` вместо `Identity` — например, таблица для `IdentityRole` называется `AspNetRoles`.

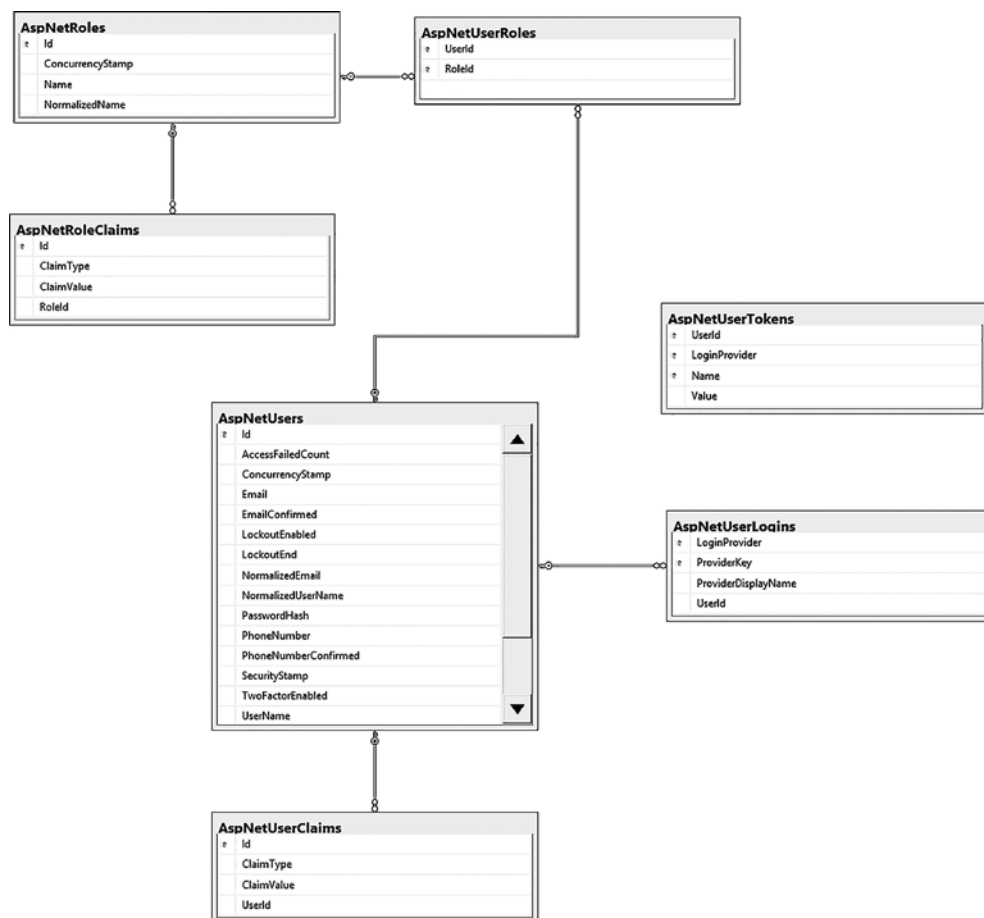


Рис. 13.6. Таблицы ASP.NET Core Identity

`IdentityUser` содержит информацию о пользователях приложения. Пользователь представляется классом `ApplicationUser`, который может расширяться так, как описано в главе 10 «Entity Framework Core». Базовый класс `IdentityUser` содержит уникальный идентификатор и ряд необязательных свойств, которые могут использоваться или не использоваться в зависимости от потребностей приложения. Например, если пользователь создает пароль в приложении, то свойство `PasswordHash` содержит криптографически сильный хеш пароля. Также имеются свойства для хранения адреса электронной почты (`Email`) и телефона

(PhoneNumber) пользователя, а также логические признаки подтверждения Email и PhoneNumber. Также существует параметр для отслеживания количества неудачных попыток входа и признак блокировки учетной записи.

ХРАНЕНИЕ ПАРОЛЕЙ

Задача хранения паролей всегда была нетривиальной. Есть несколько способов хранения паролей, и почти все они неправильны. Шифрование паролей поначалу кажется хорошей идеей, но если произойдет утечка зашифрованных паролей (а это случается с пугающей регулярностью), атакующий может расшифровать пароли. Это крайне неприятно, потому что многие люди используют одинаковые пароли на разных сайтах. Вы создаете угрозу не только для своего сайта, но и для других. С паролем и адресом электронной почты злоумышленник может войти на веб-сайт банка или другой критичный сайт. Именно по этой причине рекомендуется использовать разные пароли на разных сайтах.

Хеширование паролей — намного более лучшая идея. Хеши-функция выполняет одностороннее преобразование, результат которого не может использоваться для восстановления пароля. При это важно не только хешировать пароль пользователя, но и объединить его с солью (salt) — строкой случайных символов, присоединяемой к паролю перед хешированием. Случайная строка защищает от атак на основе радужных таблиц (rainbow tables) — больших баз данных, в которых хранятся соответствия между паролями и генерируемыми на их основе хеш-кодами. У многих популярных алгоритмов хеширования существуют радужные таблицы, которые позволяют почти мгновенно исследовать большой процент пространства ключей. С солью этот способ не сработает, потому что ключам в радужных таблицах соответствует комбинация «пароль + соль». Не зная соль, атакующий не сможет ввести работающий пароль. Вряд ли ключи в радужной таблице будут соответствовать фактическому паролю пользователя — с большой вероятностью это будет просто строка с тем же хешем, поэтому пароль пользователя не очевиден.

Даже при хешировании паролей необходимо обращать пристальное внимание на реализацию. Многие распространенные алгоритмы хеширования — такие, как MD5 и SHA, — проектировались с расчетом на максимальную скорость работы. Они предназначены прежде всего для вычисления контрольной суммы файлов с целью проверки их правильности. При хешировании больших объемов данных операция должна быть по возможности простой и быстрой. При хешировании паролей ставятся противоположные цели. Хеширование должно занимать значительное время для предотвращения атак методом «грубой силы» — желательно за счет применения настраиваемого алгоритма, который можно легко удлинить по мере удешевления вычислительных ресурсов.

Насколько просто взламывается пароль, хешированный с применением примитивной хеш-функции? Известный хакер Кевин Митник дает подсказку:

Обожаю мой новый взломщик паролей для 4 GPU. Более 60 МИЛЛИАРДОВ хешей NTLM в секунду :-). К сожалению, Md5crypt обрабатывается намного медленнее. Нужно добавить новые карты.

Кевин Митник (@kevinmitnick)

В статье «Атаки на пароли с хешированием MD5»¹, которую опубликовали в 2005 году Робертелло (Robertiello) и Бандла (Bandla), утверждалось, что пароли MD5 могут быть взломаны за 96 часов с применением простейшей реализации «грубой силы» на обычном процессоре. С современными графическими процессорами в сочетании с дешевым оборудованием — и даже эластичными вычислениями — взлом MD5 становится не только возможным, но и простым.

К счастью, все эти проблемы решает за вас ASP.NET Identity. При настройке на использование локальных учетных записей ASP.NET хеширует пароли с использованием PasswordHasher. По умолчанию PasswordHasher из ASP.NET использует алгоритм PBKDF2 (HMAC-SHA256), 128-разрядную соль, 256-разрядный подключ и 10 000 итераций. По всем критериям это очень сильный хеш, на взлом которого потребуется значительное время. Этот хеш хранится в столбце PasswordHash таблицы AspNetUsers table и используется классом SignInManager для подтверждения пароля при попытке входа.

IdentityRole представляет собой роль, а сущности **IdentityUser** могут принадлежать любому количеству классов **IdentityRole**. Принадлежность к **IdentityRole** может использоваться для авторизации пользователей областей или ресурсов приложения (см. раздел «Включение средств безопасности с использованием атрибутов» далее в этой главе).

В ASP.NET Identity интенсивно используются утверждения (claims). Утверждение представляет собой пару «имя — значение» с некоторой информацией о пользователе. Утверждение может выдаваться доверенной стороной (например, Facebook) или же вашим приложением. Примерами утверждений могут послужить адрес электронной почты или дата рождения пользователя. В ASP.NET Core Identity утверждения для конкретного пользователя представляются сущностью **IdentityUserClaim**. Утверждения также могут добавляться для группы пользователей в сущности **IdentityRoleUserClaim**, хотя эта возможность используется реже. Утверждения составляют важный аспект авторизации, как будет показано в разделе «Специальные политики авторизации» этой главы.

Сущность **IdentityUserLogin** представляет собой вход с доверенной третьей стороны. Сторонние провайдеры аутентификации (например, Facebook и Twitter) представлены в разделе «Сторонние провайдеры аутентификации».

Сущность **IdentityUserToken** представляет собой маркер доступа, относящийся к конкретному пользователю. Эти маркеры обычно используются для получения доступа к открытому API приложения. Пример такого встречается на GitHub, где пользователи могут генерировать персональные маркеры доступа для обращения к GitHub API. В приложении AlpineSkiHouse сущности **IdentityUserToken** не используются.

¹ http://pamsusc.googlecode.com/svn-history/r38/CSCI555/thesis/papers/RAINBOW_report.pdf.

Использование ASP.NET Identity

ASP.NET Core Identity автоматически добавляется в приложения, создаваемые с использованием шаблонов ASP.NET Core по умолчанию. Однако важно понимать, на какие пакеты создаются ссылки и как они настраиваются.

При создании проекта `AlpineSkiHouse.Web` был выбран вариант `Authentication: Individual User Accounts`. При выборе этого варианта добавляется ссылка на пакет `Microsoft.AspNetCore.Identity.EntityFrameworkCore`, который в свою очередь содержит ссылку на пакет `Microsoft.AspNetCore.Identity`. Шаблон проекта также генерирует заготовку кода для управления пользователями в приложении. Класс `ApplicationUser` и связанный с ним класс `DbContext` уже упоминались в главе 10.

В классе `Startup` встречается следующая конфигурация, которая сообщает ASP.NET Identity, что вы используете класс `ApplicationUser` и что для хранения модели Identity используется Entity Framework:

Конфигурация ASP.NET Identity в `Startup.ConfigureServices`

```
services.AddIdentity<ApplicationUser, IdentityRole>()
    .AddEntityFrameworkStores<ApplicationUserContext>()
    .AddDefaultTokenProviders();
    .AddEntityFrameworkStores<ApplicationUserContext>()
    .AddDefaultTokenProviders();
```

Также можно задать дополнительные параметры. Например, можно изменить количество недействительных попыток входа, после которых учетная запись автоматически блокируется:

```
services.AddIdentity<ApplicationUser, IdentityRole>(
    options =>
    {
        options.Lockout.MaxFailedAccessAttempts = 3;
    })
    .AddEntityFrameworkStores<ApplicationUserContext>()
    .AddDefaultTokenProviders();
```

Другие параметры позволяют задать требования к силе пароля, требования к пользователю, пути для входа и выхода пользователя, период времени, в течение которого cookie-файл аутентификации остается действительным, и многие другие аспекты.

СМ. ТАКЖЕ

Полный список параметров доступен по адресу <https://docs.asp.net/en/latest/security/authentication/identity.html>, а репозиторий Identity — по адресу <https://github.com/aspnet/Identity/blob/dev/src/Microsoft.AspNetCore.Identity/IdentityOptions.cs>.

Настройка конфигурации ASP.NET Identity завершается вызовом `app.UseIdentity();` в методе `Configure` класса `Startup`. Этот вызов должен располагаться непосредственно перед `app.UseMvc();`.

Учетные записи локальных пользователей

ASP.NET Identity предоставляет средства управления и аутентификации пользователей в классах `UserManager` и `SignInManager` соответственно. Шаблоны проектов по умолчанию всегда генерируют значительный объем кода для управления локальными учетными записями. Потратим немного времени для более подробного анализа этого кода.

Управление пользователями

Класс `userManager` предназначен для управления пользователями в ASP.NET Identity Core. Например, при регистрации новый пользователь создается вызовом метода `CreateAsync`.

```
var user = new ApplicationUser
{
    UserName = model.Email,
    Email = model.Email,
    FirstName = model.FirstName,
    LastName = model.LastName,
    PhoneNumber = model.PhoneNumber
};
var result = await _userManager.CreateAsync(user, model.Password);
```

Класс `UserManager` предоставляет значительную функциональность, реализовать которую вручную было бы в высшей степени утомительно. Это и механизмы подтверждения адресов электронной почты и телефонов, и сброс паролей, и установление/снятие блокировки учетных записей пользователя. Эти механизмы хорошо документированы на официальном сайте документации ASP.NET по адресу <https://docs.asp.net/en/latest/security/authentication/accconfirm.html>.

Аутентификация пользователей

Когда пользователь выбирает вариант входа с использованием локальной учетной записи, необходимо проверить пользователя по имени и паролю, введенным в форму. Проверка имени пользователя и пароля осуществляется классом `SignInManager`, внедренным в `AccountController`.

```
//
// POST: /Account/Login
[HttpPost]
```

```
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Login(LoginViewModel model, string returnUrl =
    null)
{
    ViewData["ReturnUrl"] = returnUrl;
    if (ModelState.IsValid)
    {
        var result = await _signInManager.PasswordSignInAsync(model.Email, model.
            Password, model.RememberMe, lockoutOnFailure: false);
        if (result.Succeeded)
        {
            _logger.LogInformation(1, "User logged in.");
            return RedirectToLocal(returnUrl);
        }
        if (result.RequiresTwoFactor)
        {
            return RedirectToAction(nameof(SendCode), new { ReturnUrl = returnUrl,
RememberMe = model.RememberMe });
        }
        if (result.IsLockedOut)
        {
            _logger.LogWarning(2, "User account locked out.");
            return View("Lockout");
        }
        else
        {
            ModelState.AddModelError(string.Empty, "Invalid login attempt.");
            return View(model);
        }
    }
    // Если управление перешло в эту точку, возникли проблемы;
    // форму следует отобразить заново.
    return View(model);
}
```

Если комбинация «имя пользователя/пароль» соответствует информации из базы данных, то вход считается успешным, и **SignInManager** добавляет в **HttpResponse** специальное аутентификационное значение cookie — зашифрованную строку, которая в действительности является сериализованным экземпляром **ClaimsPrincipal**. Класс **ClaimsPrincipal** однозначно идентифицирует пользователя и его утверждения. В последующих запросах ASP.NET Identity проверяет cookie-файл аутентификации, расшифровывает объект **ClaimsPrincipal** и назначает его свойству **User** объекта **HttpContext**. Таким образом, каждый запрос связывается с конкретным пользователем без необходимости передачи имени пользователя и пароля с каждым запросом. Расшифрованный и десериализованный объект **ClaimsPrincipal** доступен из действия контроллера через свойство **User**.

ПРИМЕЧАНИЕ

Механизм аутентификации на основе cookie в ASP.NET Core Identity реализуется с использованием промежуточного ПО. Для cookie такое промежуточное ПО может использоваться независимо от механизма ASP.NET Identity через пакет Microsoft.AspNetCore.Authentication.Cookies. За дополнительной информацией обращайтесь по адресу <https://docs.asp.net/en/latest/security/authentication/cookie.html>.

Класс `SignInManager` предоставляет ряд дополнительных возможностей. Например, можно решить, должна ли неудачная попытка входа учитываться для блокировки учетных записей. Блокировка учетных записей после определенного количества неудачных попыток — важная мера безопасности. Шаблон по умолчанию отключает это поведение, потому что в него не включен механизм, позволяющий системным администраторам снимать блокировку с учетных записей. Запомним это и добавим функциональность блокировки в будущем. А пока проблемы поддержки, связанные с блокировкой, передаются в группу разработки для ручного снятия блокировки в базе данных. Другой вариант — включение двухфакторной аутентификации. При включении двухфакторной аутентификации (или 2FA) после ввода действительного имени пользователя и пароля система отправляет в сообщении электронной почты или текстовом сообщении случайный код. Пользователь перенаправляется на страницу, на которой он вводит этот код. Процедура входа завершается только после того, как этот код будет подтвержден.

Полное руководство по подключению двухфакторной аутентификации с использованием SMS доступно на официальном сайте документации ASP.NET по адресу <https://docs.asp.net/en/latest/security/authentication/2fa.html>.

Сторонние провайдеры аутентификации

ASP.NET Identity предоставляет простую интеграцию со сторонними провайдерами аутентификации, которые называются внешними провайдерами входа (external login providers). Команда Alpine приняла решение реализовать возможность входа с использованием в качестве внешнего источника Twitter или Facebook (рис. 13.7).

Одной из целей проекта Parsley было расширение вовлечения пользователей через социальные сети, поэтому выбор Facebook и Twitter в качестве провайдеров аутентификации для приложения был естественным.

Чтобы разрешить вход с Facebook и Twitter, добавьте ссылки на соответствующие пакеты NuGet.

```
"Microsoft.AspNetCore.Authentication.Facebook": "1.0.0",  
"Microsoft.AspNetCore.Authentication.Twitter": "1.0.0"
```

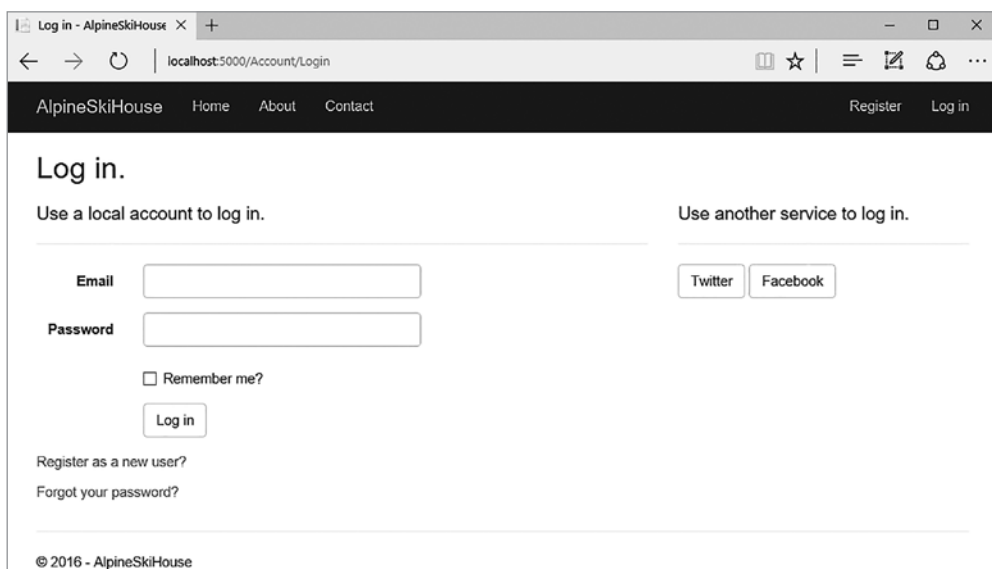


Рис. 13.7. Вход с внешним провайдером входа

Затем настройте провайдеров в методе `Configure` класса `Startup`.

```
app.UseIdentity();

if (Configuration["Authentication:Facebook:AppId"] == null ||
    Configuration["Authentication:Facebook:AppSecret"] == null ||
    Configuration["Authentication:Twitter:ConsumerKey"] == null ||
    Configuration["Authentication:Twitter:ConsumerSecret"] == null)
    throw new KeyNotFoundException("A configuration value is missing for
    authentication against Facebook and Twitter. While you don't need to get tokens for
    these you do need to set up your user secrets as described in the readme.");
app.UseFacebookAuthentication(new FacebookOptions
{
    AppId = Configuration["Authentication:Facebook:AppId"],
    AppSecret = Configuration["Authentication:Facebook:AppSecret"]
});

app.UseTwitterAuthentication(new TwitterOptions
{
    ConsumerKey = Configuration["Authentication:Twitter:ConsumerKey"],
    ConsumerSecret = Configuration["Authentication:Twitter:ConsumerSecret"]
});
```

Каждому внешнему провайдеру входа требуется набор открытых и частных маркеров. Метод получения маркеров зависит от провайдера; за конкретными

инструкциями обращайтесь к документации разработчика, предоставляемой провайдером. Как упоминалось ранее в разделе «Внутренние угрозы» этой главы, эти маркеры должны оставаться недоступными для внешнего доступа, поэтому их необходимо читать из раздела конфигурации, вместо того чтобы встраивать в код.

На странице входа создаются кнопки для входа с любого из настроенных внешних провайдеров. Когда пользователь выбирает одного из этих провайдеров (например, провайдера Facebook), он перенаправляется на страницу входа Facebook, где ему предлагается ввести имя пользователя и пароль Facebook. Если предположить, что вход Facebook прошел успешно, пользователь перенаправляется на конечную точку `Account/ExternalLoginCallback` веб-приложения Alpine Ski House. В этом методе действия приложение получает информацию о пользователе от `SignInManager`, после чего пытается выполнить вход с использованием внешней информации.

```
ExternalLoginInfo info = await _signInManager.GetExternalLoginInfoAsync();
if (info == null)
{
    return RedirectToAction(nameof(Login));
}

// выполнить вход с внешним провайдером.
var result = await _signInManager.ExternalLoginSignInAsync(info.LoginProvider,
    info.ProviderKey, isPersistent: false);
if (result.Succeeded)
{
    _logger.LogInformation(5, "User logged in with {Name} provider.", info.
LoginProvider);
    return RedirectToLocal(returnUrl);
}
if (result.RequiresTwoFactor)
{
    return RedirectToAction(nameof(SendCode), new { ReturnUrl = returnUrl });
}
if (result.IsLockedOut)
{
    return View("Lockout");
}
```

Если вход не был успешным, необходимо предоставить пользователю возможность создания новой учетной записи в Alpine Ski House. При создании новой учетной записи необходимо сохранить дополнительную информацию о пользователе: адрес электронной почты, имя, фамилию, телефон и возраст.

Некоторые внешние провайдеры входа предоставляют часть этой информации в виде значений утверждений, которые находятся в свойстве `Principal` объекта `ExternalLoginInfo`. Например, провайдер Facebook предоставляет адрес электронной почты, имя и фамилию пользователя; эти данные используются для предварительного заполнения данных пользователя. Провайдер Twitter эти данные не предоставляет.

```
// Если у пользователя нет учетной записи,  
// предложить пользователю создать ее.  
ViewData["ReturnUrl"] = returnUrl;  
ViewData["LoginProvider"] = info.LoginProvider;  
var email = info.Principal.FindFirstValue(ClaimTypes.Email);  
var lastName = info.Principal.FindFirstValue(ClaimTypes.Surname);  
var firstName = info.Principal.FindFirstValue(ClaimTypes.GivenName);  
  
return View("ExternalLoginConfirmation",  
    new ExternalLoginConfirmationViewModel  
    {  
        Email = email,  
        FirstName = firstName,  
        LastName = lastName  
    });
```

ПРИМЕЧАНИЕ

Компания Microsoft предоставляет пакеты, интегрируемые со встроенными конструкциями идентификации, для поддержки аутентификации через учетные записи Facebook, Twitter, Google и Microsoft; все эти пакеты доступны через NuGet. Также возможна поддержка входа через других провайдеров с помощью специальных реализаций, написанных вами или третьими сторонами. Обучающее руководство по использованию аутентификации GitHub доступно по адресу <http://aspnetmonsters.com/2016/04/github-authentication-asp-net-core/>.

Включение средств безопасности с использованием атрибутов

Одно из преимуществ создания приложений на базе полнофункционального фреймворка заключается в том, что многие необязательные компоненты хорошо дополняют друг друга. После включения аутентификации в приложения с использованием промежуточного ПО вы сможете пользоваться ей при помощи глобальных фильтров и атрибутов, которыми помечаются ваши классы и методы. Фильтры будут рассматриваться в следующем разделе, но эта возможность может быть знакома любому разработчику в области MVC. В ASP.NET Core они реализуются на базе концепции политик приложений.

В простейшем варианте, для того чтобы сообщить о необходимости аутентификации пользователя, достаточно пометить класс атрибутом **Authorize**, как в следующем определении **SkiCardController**:

```
[Authorize]  
public class SkiCardController : Controller  
{  
    // ...  
}
```

Если же вы не хотите, чтобы каждое действие требовало аутентификации, воспользуйтесь атрибутом на уровне метода:

```
[Authorize]
public async Task<ActionResult> Create()
{
    // ...
}
```

Если вы пометили свой контроллер атрибутом `Authorize`, возможно, вы захотите предоставить пользователю, не прошедшему аутентификацию, доступ к находящемуся в нем действию. Обычно эта возможность используется в `AccountController` и других похожих классах, которые требуют аутентификации пользователя, но в процессе регистрации у пользователя еще нет учетной записи для входа. Для этого действие помечается следующим атрибутом:

```
[AllowAnonymous]
public IActionResult Register(string returnUrl = null)
{
    ViewData["ReturnUrl"] = returnUrl;
    return View();
}
```

Атрибут можно дополнительно уточнить и задать роли пользователя, прошедшего аутентификацию; для этого следует передать значения конструктору атрибута: `[Authorize(Roles = "customer,csr")]`. Если вы используете один атрибут, пользователь должен принадлежать хотя бы одной роли из списка, разделенного запятыми. С несколькими атрибутами `Authorize` он должен принадлежать всем перечисленным ролям.

Если вы добавите атрибут `Authorize` с указанием роли, то фреймворк постарается разрешить эту роль с `RolesAuthorizationRequirement`. Он использует метод `ClaimsPrincipal.IsInRole()` для перебора всех ролей, к которым принадлежит пользователь в соответствии с его утверждениями. По сути, если процесс входа каким-либо образом добавит пользователю утверждение, оно будет добавлено в сертификат пользователя, а пользователь получит доступ к соответствующим частям вашего приложения. Впрочем, возможности этого механизма ограничены, потому что задаваемые роли фиксируются во время компиляции и не могут изменяться во время выполнения.

К счастью, никакого волшебства в этом нет, и атрибут `Authorize` — не единственный вариант. У вас есть возможность строить для определения безопасности типы конструкций, которые подходят для вашей организации. Вместо того чтобы использовать константы в своем коде, вы можете строить расширенные требования для защиты вашего приложения, о чем будет рассказано ниже.

Применение политик для авторизации

Политика (policy) представляет собой набор требований для управления доступом, зависящих от свойств запроса, клиента, выдавшего запрос, утверждений относительно пользователя, инициировавшего запрос, и любых других факторов, которые вы захотите учесть. В исходном состоянии можно создавать политики, включающие требования на основании операций, ролей и других утверждений.

Глобальное применение политик

Глобальное требование аутентификации пользователя — распространенная практика; каждый, кто хочет обратиться к сайту, должен предоставить некий маркер идентификации. В большинстве случаев эта проблема решается браузером на стороне клиента и промежуточным ПО на стороне сервера. Решение использует cookie и согласовывается через модель «запрос/вход». Включение глобального правила обязательной аутентификации для любых обращений к приложению требует создания политики и ее добавления в коллекцию `FilterCollection` приложения. В приложении `Alpine Ski House` эта возможность не реализована, но если бы мы решили ее реализовать, это делалось бы во время запуска приложения. В метод `ConfigureServices` файла `Startup.cs` входит строка, добавляющая сервисы MVC в контейнер внедрения зависимостей:

```
services.AddMvc();
```

У `AddMvc()` также есть перегруженная версия, которая получает действие с параметром `MvcOptions`. `MvcOptions` предоставляет ряд свойств, в одном из которых содержится коллекция фильтров, используемая в приложении `Alpine Ski House`. Фильтр создается и добавляется в коллекции следующим образом:

```
services.AddMvc(options =>
{
    var policy = new AuthorizationPolicyBuilder()
        .RequireAuthenticatedUser()
        .Build();
    options.Filters.Add(new AuthorizeFilter(policy));
});
```

При этом на заднем плане происходит то же самое, что при добавлении атрибута: требование добавляется в конвейер типа `DenyAnonymousAuthorizationRequirement`. При выполнении запроса это требование нарушается для пользователя, не выполнившего вход. При необходимости фильтр можно обойти, используя атрибут `AllowAnonymous` на уровне контроллеров или действий в вашем приложении.

Определение специализированных политик

Не стоит думать, что класс `AuthorizationPolicyBuilder` умеет решать только одну задачу. С его помощью можно строить политики, которые:

- Требуют определенных утверждений.
- Требуют конкретных значений утверждений.
- Требуют конкретных ролей.
- Используют конкретные схемы аутентификации.
- Объединяют любые из этих вариантов в одном правиле.

А теперь посмотрим, что необходимо для определения и использования административной политики в нашем приложении. Требования к политике могут быть достаточно тривиальными. Например, необходимо убедиться в том, что проверенные пользователи работают в отделе кадров или бухгалтерии и обладают административной ролью в приложении. Если утверждениям ролей и отделов были назначены правильные атрибуты, определение политики будет выглядеть так:

```
var policy = new AuthorizationPolicyBuilder()
    .RequireAuthenticatedUser()
    .RequireRole("admin")
    .RequireClaim("department", new List<string> { "hr", "accounting" })
    .Build();
```

Это определение располагается в классе `Startup` метода `ConfigureServices`. После того как политика будет определена, она добавляется в набор сервисов, доступных в приложении:

```
services.AddAuthorization(options =>
{
    options.AddPolicy("RefundAccess", policy);
});
```

И хотя класс `AuthorizationPolicyBuilder` с его гибким синтаксисом весьма удобен, также возможно определить политики во встроенном формате с лямбда-переопределением:

```
services.AddAuthorization(options =>
{
    options.AddPolicy("RefundAccess", policy =>
    {
        policy.RequireAuthenticatedUser();
        policy.RequireRole("admin");
        policy.RequireClaim("department", new List<string> { "hr", "accounting" });
    });
});
```

После создания именованной политики `RefundAccess` теперь можно помечать любые контроллеры и действия атрибутом `Authorize`, как и прежде, но указывая при этом имя политики:

```
[Authorize("RefundAccess")]
public class HomeController : Controller
{
    // ...
}
```

Эти средства достаточны для многих ситуаций авторизации в приложении, но у них тоже есть ограничения. Например, политики, созданные с использованием `AuthorizationPolicyBuilder` или лямбда-методов, упоминавшихся ранее, фактически остаются неизменными на протяжении жизненного цикла приложения, потому что они настраиваются при запуске. Кроме того, эти методы не позволяют условиям, возникающим во время выполнения, влиять на принятие или отклонение пользователя политикой. Если требования и политики, доступные во фреймворке, не предоставляют нужной степени контроля, вы можете создавать и интегрировать собственные реализации.

Специальные политики авторизации

Атрибуты авторизации и политики утверждений предоставляют большую часть функциональности, необходимой для авторизации в приложении *Alpine Ski House*, но в некоторых случаях для принятия решения о том, разрешено ли пользователю сделать что-либо в приложении, требуется реализация нестандартной логики. В ASP.NET Core Identity эта логика определяется в специальных политиках авторизации. Политика авторизации состоит из одного или нескольких требований и обработчика, который проверяет эти требования в текущем контексте авторизации.

Требование (*requirement*) представляет собой обычный класс со свойствами, определяющими требование. Класс требования должен наследовать от `IAuthorizationRequirement`. Гипотетическое требование в следующем примере состоит в том, что работник должен проработать в компании более заданного периода времени.

```
public class EmploymentDurationRequirement : IAuthorizationHandler
{
    public EmploymentDurationRequirement(int minimumMonths)
    {
        MinimumMonths = minimumMonths;
    }

    public int MinimumMonths { get; set; }
}
```

Логика проверки требований политики реализуется в обработчике авторизации. Обработчики авторизации наследуют от базового класса `Authorization<T>`, где `T` — тип проверяемого требования. Например, обработчик `EmploymentDurationRequirement` может проверить минимальную продолжительность работы в месяцах на основании утверждения с датой поступления на работу.

```
public class MinimumEmploymentDurationHandler :
    AuthorizationHandler<EmploymentDurationRequirement>
{
    protected override Task HandleRequirementAsync(AuthorizationContext context,
        EmploymentDurationRequirement requirement)
    {
        if (!context.User.HasClaim(c => c.Type == ClaimTypes.EmploymentDate &&
            c.Issuer == "http://alpineskihouse.com"))
        {
            return Task.FromResult(0);
        }

        var employmentDate = Convert.ToDateTime(context.User.FindFirst(
            c => c.Type == ClaimTypes.EmploymentDate &&
            c.Issuer == "http://alpineskihouse.com").Value);

        int numberOfMonths = (DateTime.Today - employmentDate).TotalMonths;
        if (numberOfMonths > requirement.MinimumMonths)
        {
            context.Succeed(requirement);
        }
        return Task.FromResult(0);
    }
}
```

После определения требований и обработчиков специальная политика может определяться в методе `ConfigureServices` в `Startup.cs`. Значение cookie также должно быть зарегистрировано в коллекции `services`.

```
services.AddAuthorization(options =>
{
    options.AddPolicy("CompletedProbation",
        policy => policy.Requirements.Add(new MinimumEmploymentDurationHandler
            (3)));
});
services.AddSingleton<IAuthorizationHandler, MinimumEmploymentDurationHandler>();
```

Специальные политики используются так же, как и любые другие политики авторизации.

```
[Authorize(Policy = "CompletedProbation")]
public IActionResult GetSomeFreeSwag()
{
    return View();
}
```

Защита ресурсов

Часто бывает необходимо защитить доступ к некоторому ресурсу на основании логики авторизации. Например, объект `SkiCard` может редактироваться только клиентом, создавшим этот объект. В нашей исходной реализации эта проблема решалась логикой метода действия `Edit` класса `SkiCardController`. Добавляя в запрос `SkiCard` условие `s.ApplicationUserId == userId`, мы можем проследить за тем, чтобы клиенты могли редактировать только свои объекты `SkiCard`.

```
var skiCard = await _skiCardContext.SkiCards
    .SingleOrDefaultAsync(s => s.ApplicationUserId == userId
        && s.Id == viewModel.Id);

if (skiCard == null)
{
    return NotFound();
}
// Иначе продолжается действие Edit
```

Такое решение работает для простых сценариев, но оно недостаточно надежно, а его возможности ограничены. Ограничение состоит в том, что только владелец ресурса `SkiCard` может редактировать этот ресурс `SkiCard`, и не выражено явным образом. Кроме того, в приложении доверенным системным администраторам должно быть разрешено редактирование данных карт. С включением такого уровня сложности в запрос теряется ясность логики.

Для упрощения процесса авторизации можно создать специальный обработчик `AuthorizationHandler` для сущности `SkiCard`. Сначала необходимо описать реализацию `IAuthorizationRequirement`, определяющую тип требования для обработчика. В данном случае редактируются сущности `SkiCard`, поэтому создается класс с именем `EditSkiCardAuthorizationRequirement`.

```
public class EditSkiCardAuthorizationRequirement : IAuthorizationRequirement
{
}
```

Затем создайте класс `EditSkiCardAuthorizationHandler`, наследующий от базового класса `AuthorizationHandler`; тип требования и тип ресурса передаются в обобщенных параметрах. В методе `HandleRequirementAsync` добавьте логику проверки того, разрешено ли текущему пользователю редактировать заданную сущность `SkiCard`.

```
public class EditSkiCardAuthorizationHandler :
    AuthorizationHandler<EditSkiCardAuthorizationRequirement, SkiCard>
{
    private readonly UserManager<ApplicationUser> _userManager;

    public EditSkiCardAuthorizationHandler(UserManager<ApplicationUser>
        userManager)
```

```

    {
        _userManager = userManager;
    }

    protected override Task HandleRequirementAsync(AuthorizationHandlerContext
        context, EditSkiCardAuthorizationRequirement requirement, SkiCard
        skiCard)
    {
        var userId = _userManager.GetUserId(context.User);
        if (skiCard.ApplicationUserId == userId)
        {
            context.Succeed(requirement);
        }
        return Task.CompletedTask;
    }
}

```

Теперь требования к авторизации для редактирования `SkiCard` четко выражены и пригодны для тестирования (см. главу 20 «Тестирование»). Далее зарегистрируйте `AuthorizationHandler` в методе `ConfigureServices` в `Startup.cs`.

```
services.AddSingleton<IAuthorizationHandler, EditSkiCardAuthorizationHandler>();
```

Наконец, измените `SkiCardController` для правильной авторизации пользователей в методе действия `Edit`. В отличие от специальной политики авторизации, упоминавшейся выше, декларативный подход не может использоваться с атрибутом `Authorize`, потому что ресурс `SkiCard` не будет известен на момент выполнения атрибута `Authorize`. Вместо этого используйте `IAuthorizationService` для проведения императивной авторизации.

```

private readonly SkiCardContext _skiCardContext;
private readonly UserManager<ApplicationUser> _userManager;
private readonly IAuthorizationService _authorizationService;

public SkiCardController(SkiCardContext skiCardContext,
    UserManager<ApplicationUser> userManager,
    IAuthorizationService authorizationService)
{
    _skiCardContext = skiCardContext;
    _userManager = userManager;
    _authorizationService = authorizationService;
}

```

В методе действия `Edit` вызовите метод `AuthorizeAsync` сервиса авторизации, передавая ему текущего пользователя, карту и экземпляр `EditSkiCardAuthorizationRequirement`. Если метод `AuthorizeAsync` возвращает `true`, можно продолжать. Если нет — возвращается объект `ChallengeResult`, который перенаправляет пользователя на `Account/AccessDenied`.

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Edit(EditSkiCardViewModel viewModel)
{
    if (ModelState.IsValid)
    {
        var skiCard = await _skiCardContext.SkiCards
            .SingleOrDefaultAsync(s => s.Id == viewModel.Id);

        if (skiCard == null)
        {
            return NotFound();
        }
        else if (await _authorizationService.AuthorizeAsync(User, skiCard, new
            EditSkiCardAuthorizationRequirement()))
        {
            skiCard.CardHolderFirstName = viewModel.CardHolderFirstName;
            skiCard.CardHolderLastName = viewModel.CardHolderLastName;
            skiCard.CardHolderPhoneNumber = viewModel.CardHolderPhoneNumber;
            skiCard.CardHolderBirthDate = viewModel.CardHolderBirthDate.Value.Date;

            await _skiCardContext.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
        else
        {
            return new ChallengeResult();
        }
    }
    return View(viewModel);
}
```

Общий доступ к ресурсам независимо от источника (CORS)

Для отображения полнофункциональной веб-страницы необходимы многие типы контента и поведения, от изображений и таблиц стилей до сценариев. Частью такого контента могут быть данные или запросы AJAX, обладающие возможностью подгружать дополнительную информацию для браузера. Чтобы такие запросы не были перехвачены злоумышленниками, комитет W3C (World Wide Web Consortium) встроил в рекомендуемые стандарты предложение использовать CORS (Cross-Origin Resource Sharing) для установления доверия между клиентом и сервером.

Для работы этого механизма браузер, выдающий запрос с использованием AJAX к любому домену, отличному от источника самой веб-страницы, должен представить заголовок `Origin`. Аналогичным образом сервер должен ответить заголовком `Access-Control-Allow-Origin`, чтобы сообщить, что он ожидает запрос из этого

домена. Если ожидаемые значения или заголовок не будут получены, то браузер запрещает сценарию, запросившему ресурс, обращаться к ответу, полученному от сервера. Хотя такая схема не защищает клиента в случае нарушения системы безопасности на стороне сервера, она устраняет большую часть рисков межсайтовых сценарных атак.

Если вы намерены открыть API к сайту, которые происходят из разных источников, в приложении необходимо включить поддержку CORS. Два запроса происходят из одного источника, если они оба работают на одном домене, одном порте и одной схеме (HTTP или HTTPS). CORS разумно включать в том случае, если конкретные источники можно изменять или объявлять недействительными «на лету». Пример такого рода — API, потребляемый третьими сторонами, которые платят за доступ к вашему API.

Чтобы включить поддержку CORS в приложении, необходимо сначала добавить пакет `Microsoft.AspNetCore.Cors` в `project.json`. Затем необходимо добавить политику CORS в контейнер сервисов в `Startup.cs` через `CorsPolicyBuilder`:

```
public void ConfigureServices(IServiceCollection services)
{
    // Конфигурация других сервисов
    services.AddCors(options =>
    {
        options.AddPolicy("AlpineSkiHouseApiPolicy",
            builder => builder.WithOrigins("https://alpineskihouse.com"));
    });
    // ...
}
```

Вызов `AddPolicy()` позволяет выбрать имя и построить нужную политику. Затем промежуточное ПО связывается с промежуточным ПО CORS перед вызовом `UseMvc()` в методе `Configure` в файле `Startup.cs`:

```
app.UseCors("AlpineSkiHouseApiPolicy");
```

Возможно, эта политика будет слишком широкой для всего API, поэтому вызов `UseCors()` можно полностью опустить и вместо этого воспользоваться решением с атрибутами на уровне контроллера.

```
[EnableCors("AlpineSkiHouseApiPolicy ")]
public class SkiPassValidationApiController : Controller
{
    // ...
}
```

Этот способ позволяет подключать CORS для конкретных аспектов вашего API с более высокой детализацией. Вы также можете использовать `EnableCORS` на

уровне действий, а чтобы отключить CORS для некоторого включенного в политику действия (глобально или на уровне контроллера), примените атрибут `DisableCors`.

Итоги

Безопасность в приложениях — весьма многогранная тема, о которой написано множество книг. С учетом сегодняшних настроений в СМИ, растущей озабоченности клиентов в отношении безопасности и растущей угрозы взлома приложений необходимо делать все возможное для того, чтобы доступ к различным частям нашего приложения получали только стороны, обладающие необходимыми разрешениями.

Конечно, имеющиеся внутренние проблемы реальны, и нам как разработчикам приходится думать о безопасности при написании каждой строки кода. Но помимо того, что происходит во внутренней реализации (включая то, как пишется код и кому предоставляется доступ), также важно помнить о тех, кто пытается получить доступ к системе без нашего разрешения. За дополнительной информацией о безопасности в системах, рассчитанных на обслуживание внешних пользователей, мы рекомендуем обратиться к трудам сообщества OWASP (Open Web Application Security Project) по адресу <https://www.owasp.org/>.

Кроме того, тема безопасности весьма обширна, и хотя в этой главе мы рассмотрели более десятка проблем, в ней даже бегло не упоминаются такие традиционные концепции, как сертификаты или атаки внедрения SQL, — причем вовсе не потому, что эти темы менее важны. Безопасность безусловно должна постоянно занимать мысли каждого разработчика и оставаться лейтмотивом вашего проекта.

Безопасность — всего лишь одна категория сервисов, которые должны использоваться в вашем приложении. В следующей главе будет показано, как предоставить доступ к этим сервисам каждому компоненту вашего проекта так, чтобы этим компонентам не нужно было знать ничего конкретного о создании экземпляров этих сервисов.

14

Внедрение зависимостей

«Это не код, а какая-то мешанина, — пожаловался Марк-2. — Понятия не имею, как это все тестировать. На самом деле мне нужно протестировать всего один приватный метод, но до него трудно добраться. Может, его стоит просто сделать публичным».

«Давай посмотрим, — ответила Даниэль, — как называется этот твой метод?»

«CalculatePassStartDate».

«В каком классе?»

«PassValidationChecker».

«Хм, — сказала Даниэль, — Мне кажется, что этот класс делает слишком много всего. Для чего он — для проверки абонементов или вычисления даты начала его действия?»

«И то и другое. Дата начала нужна проверка действительности абонемента, так что логично, что эта функциональность совмещается в одном классе».

«Хорошо, и какой класс нужно будет изменять при добавлении нового типа абонемента?»

«Получается, что PassValidationChecker».

«А какой класс нужно будет изменять при добавлении нового курорта?»

«Снова PassValidationChecker. Там хранится список правил для каждого склона. Там же придется вносить изменения при изменении такого правила, как возраст, в котором перестает действовать скидка на детские абонементы».

«Выходит, у класса как минимум две причины для изменения... а скорее всего, больше. Это признак того, что классов должно быть несколько. Вспомни принципы SOLID — буква S означает "единственную ответственность" (Single Responsibility). Класс должен быть настолько узкоспециализированным, чтобы он делал что-то одно, и поэтому была только одна причина для его изменения».

«Очко в твою пользу, — сказал Марк-2. — Значит, нужно извлечь метод `CalculatePassStartDate`, скорее всего в классе `PassStartDateCalculator`, и затем создать его экземпляр в `PassValidationChecker`».

«Постой, что значит создать экземпляр?»

«Как обычно, конструкцией `new PassStartDateCalculator()`».

«Но это никак не решит существующую проблему связанности. Класс все равно будет трудно изолировать для тестирования, потому что мы зависим от конкретной реализации».

«Даниэль, его придется создать. Он же не появится по волшебству в `PassValidationChecker`».

«Может появиться, если ты добавишь его в наш контейнер внедрения зависимостей и воспользуешься интерфейсом».

«О, хорошая идея, Даниэль. Сильно напоминает инверсию управления — букву I в SOLID».

«Да, внедрение зависимостей — один из простых способов реализации инверсии управления. В .NET Core ничего проще и быть не может. Встроенный контейнер уже существует, тебе остается только зарегистрировать в нем свою реализацию».

«Спасибо, Даниэль. Сразу видно SOLID'ного программиста», — сказал Марк-2, усмехаясь.

Даниэль закатила глаза. Дважды. «Проваливай, Марк».

Эта глава не случайно упоминалась в нескольких местах книги. Внедрение зависимостей (DI, Dependency Injection) стало одним из важнейших архитектурных факторов при проектировании ASP.NET Core. Возможно, среди читателей найдутся эксперты в этой области, но большинству представленные концепции неизвестны. В этой главе мы начнем с общего описания внедрения зависимостей, а потом займемся реализацией, предоставляемой ASP.NET Core.

Что такое «внедрение зависимостей»?

Прежде чем браться за основы внедрения зависимостей, нужно понять, какую проблему пытается решить внедрение зависимостей. Обратите внимание: в контексте этой темы все контроллеры, компоненты и другие классы, составляющие наше приложение, будут называться «сервисами».

Ручное разрешение зависимостей

Для начала возьмем относительно простой класс `SkiCardController` из главы 10 «Entity Framework Core». Чтобы обрабатывать запросы на просмотр, создание

и редактирование карт, класс `SkiCardController` должен пользоваться другими сервисами нашего приложения. А именно ему нужен класс `SkiCardContext` для обращения к данным, `userManager<ApplicationUser>` для обращения к информации о текущем пользователе и `IAuthorizationService` для проверки того, авторизован ли текущий пользователь для редактирования или просмотра карты.

Без использования DI или других паттернов классу `SkiCardController` придется принять на себя ответственность за создание новых экземпляров этих сервисов.

SkiCardController без DI

```
public class SkiCardController : Controller
{
    private readonly SkiCardContext _skiCardContext;
    private readonly UserManager<ApplicationUser> _userManager;
    private readonly IAuthorizationService _authorizationService;

    public SkiCardController()
    {
        _skiCardContext = new SkiCardContext(new DbContextOptions<SkiCardContext>());
        _userManager = new UserManager<ApplicationUser>();
        _authorizationService = new DefaultAuthorizationService();
    }

    // Методы действий
}
```

Код выглядит относительно просто, но не компилируется. Прежде всего, мы не указали явно строку подключения или тип базы данных для `SkiCardContext`, так что `DbContext` создается неправильно. У `userManager<ApplicationUser>` нет конструктора по умолчанию. Единственный открытый конструктор класса `UserManager` получает семь параметров:

Открытый конструктор класса UserManager<TUser>

```
public UserManager(IUserStore<TUser> store, IOptions<IdentityOptions>
    optionsAccessor, IPasswordHasher<TUser> passwordHasher, IEnumerable<
    IUserValidator<TUser>> userValidators, IE numerable<IPasswordValidator<
    TUser>> passwordValidators, ILookupNormalizer keyNormalizer,
    IdentityErrorDescriber errors, IServiceProvider services,
    ILogger<UserManager<TUser>> logger)
{ //...
}
```

Итак, класс `SkiCardController` должен знать, как создавать все эти сервисы. Конструктор `DefaultAuthorizationService` тоже имеет три параметра. Требовать, чтобы наши контроллеры или любые другие сервисы в приложении создавали все сервисы, с которыми они взаимодействуют, невозможно.

Кроме значительного дублирования кода, неизбежного в этой реализации, такое решение также создает сильную связанность в коде. Например, `SkiCardController` теперь обладает конкретной информацией о конкретном классе `DefaultAuthorizationService` вместо общего понимания методов, предоставляемых интерфейсом `IAuthorizationService`. Если мы когда-нибудь захотим изменить конструктор `DefaultAuthorizationService`, то придется также вносить изменения в `SkiCardController` и во всех остальных классах, использующих `DefaultAuthorizationService`.

Сильная связанность также затрудняет замену реализаций. Вряд ли мы станем реализовывать совершенно новый сервис авторизации, но возможность замены реализаций важна, потому что она упрощает макетирование (*mocking*). Макетирование — важная методология, значительно упрощающая тестирование предполагаемых взаимодействий между сервисами в приложении (см. главу 20 «Тестирование»).

Использование контейнера сервисов для разрешения зависимостей

Внедрение зависимостей — стандартный паттерн, используемый для разрешения зависимостей. При внедрении зависимостей ответственность за создание и управление экземплярами класса выносится в контейнер. Кроме того, в каждом классе объявляется, от каких других классов он зависит. Контейнер может разрешать такие зависимости во время выполнения и передавать их по мере необходимости. Паттерн внедрения зависимостей является разновидностью паттерна инверсии управления (IoC, *Inversion of Control*), при котором сами компоненты уже не несут ответственности за прямое создание своих зависимостей. Реализации DI так же иногда называются «контейнерами IoC».

Самый распространенный способ внедрения зависимостей основан на приеме, называемом внедрением зависимостей через конструктор. При внедрении зависимостей через конструктор класс объявляет открытый конструктор, который получает аргументы для всех сервисов, которые ему необходимы. Например, у `SkiCardController` есть конструктор, который получает `SkiCardContext`, `UserManager<ApplicationUser>` и `IAuthorizationService`. Контейнер обеспечивает передачу экземпляров всех этих типов во время выполнения.

SkiCardController с использованием внедрения зависимостей через конструктор

```
public class SkiCardController : Controller
{
    private readonly SkiCardContext _skiCardContext;
    private readonly UserManager<ApplicationUser> _userManager;
    private readonly IAuthorizationService _authorizationService;

    public SkiCardController(SkiCardContext skiCardContext,
                            UserManager<ApplicationUser> userManager,
```

```
        IAuthorizationService authorizationService)
    {
        _skiCardContext = skiCardContext;
        _userManager = userManager;
        _authorizationService = authorizationService;
    }
    // Методы действий
}
```

Внедрение зависимостей через конструктор четко сообщает, какие зависимости необходимы тому или иному классу. Вы даже получите некоторую помощь от компилятора, потому что новый экземпляр `SkiCardController` невозможно создать без передачи необходимых сервисов. Как упоминалось ранее, одно из ключевых преимуществ такого подхода — серьезное упрощение модульного тестирования.

В другом способе внедрения зависимостей — внедрении зависимостей через свойство — открытое свойство помечается атрибутом, показывающим, что свойство должно задаваться во время выполнения контейнером. Пожалуй, внедрение зависимостей через свойство встречается реже, чем внедрение через конструктор, и поддерживается лишь частью контейнеров IoC.

Сервисы регистрируются в контейнере при запуске приложения. Способ регистрации сервисов зависит от используемого контейнера. Примеры будут приведены позже в этой главе.

ПРИМЕЧАНИЕ

В настоящее время внедрение зависимостей является самым популярным, но не единственным паттерном разрешения зависимостей. В течение некоторого времени в сообществе .NET был популярен паттерн «Локатор сервисов»; с этим паттерном сервисы регистрируются в центральном «локаторе». Каждый раз, когда сервису потребуется экземпляр другого сервиса, он запрашивает этот экземпляр у локатора. Основной недостаток этого паттерна заключается в том, что у каждого сервиса появляется явная зависимость от локатора. За подробным анализом двух паттернов обращайтесь к превосходной статье Мартина Фаулера (Martin Fowler) по теме¹.

Внедрение зависимостей в ASP.NET Core

ASP.NET Core предоставляет базовую реализацию контейнера с встроенной поддержкой внедрения через конструктор. Сервисы регистрируются во время запуска приложения в методе `ConfigureServices` класса `Startup`.

Метод `ConfigureServices` в `Startup.cs`

```
// Метод вызывается исполнительной средой. Используйте его
// для добавления сервисов в контейнер.
```

¹ <http://www.martinfowler.com/articles/injection.html>

```
public void ConfigureServices(IServiceCollection services)
{
    // Добавьте сервисы
}
```

Даже в простейшей версии проекта ASP.NET Core MVC для нормальной работы приложения в контейнер нужно будет добавить как минимум несколько сервисов. Сам фреймворк MVC зависит от сервисов, которые должны существовать во время выполнения; они необходимы для поддержки активизации контроллеров, визуализации представлений и ряда других базовых концепций.

Использование встроенного контейнера

Начать следует с добавления сервисов, предоставляемых фреймворком ASP.NET Core. При попытке вручную регистрировать все сервисы из ASP.NET Core метод `ConfigureServices` быстро выходит из-под контроля. К счастью, каждый функциональный аспект фреймворка предоставляет удобный метод расширения `Add*` для простого добавления сервисов, необходимых для включения этой возможности. Например, метод `AddDbContext` используется для регистрации `DbContext` из Entity Framework. Некоторые методы также предоставляют делегата `options`, который позволяет провести дополнительную настройку регистрируемого сервиса. Например, при регистрации классов `DbContext` при помощи делегата можно указать, что контекст должен подключиться к базе данных SQL Server, указанной в строке подключения `DefaultConnection`.

Регистрация `DbContext` в `AlpineSkiHouse.Web`

```
services.AddDbContext<ApplicationUserContext>(options =>
    options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
services.AddDbContext<SkiCardContext>(options =>
    options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
services.AddDbContext<PassContext>(options =>
    options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
services.AddDbContext<PassTypeContext>(options =>
    options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
services.AddDbContext<ResortContext>(options =>
    options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection"));
```

Также здесь добавляется `Identity` для аутентификации и авторизации, `Options` для конфигурации с сильной типизацией и, наконец, MVC для использования маршрутизации, контроллеров и всего остального, что присуще MVC.

```
services.AddIdentity<ApplicationUser, IdentityRole>()
    .AddEntityFrameworkStores<ApplicationUserContext>()
    .AddDefaultTokenProviders();
services.AddOptions();
services.AddMvc();
```

На следующем шаге регистрируются сервисы, написанные вами или взятые из сторонних библиотек. Команда Alpine Ski тоже проследила за тем, чтобы все сервисы, необходимые для контроллеров (а также все, что необходимо этим сервисам), были должным образом зарегистрированы. При регистрации сервисов приложения важно учитывать срок жизни добавляемого сервиса.

ПРИМЕЧАНИЕ

Одной из обязанностей контейнера является управление жизненным циклом сервиса. Жизненный цикл определяется как время существования сервиса, от его первого создания контейнером внедрения зависимостей до момента освобождения всех его экземпляров контейнером. Контейнер ASP.NET поддерживает варианты жизненного цикла, перечисленные в табл. 14.1.

Таблица 14.1. Варианты жизненного цикла контейнера ASP.NET

Transient	Новый экземпляр создается при каждом запросе к сервису. Этот вариант используется для компактных, легких сервисов
Scoped	Для каждого запроса HTTP создается один экземпляр
Singleton	Единственный экземпляр создается при первом запросе к сервису
Instance	Аналогичен предыдущему, но экземпляр регистрируется в контейнере при запуске

При добавлении `DbContext` методом `AddDbContext` контекст регистрируется с жизненным циклом `Scoped`. Когда запрос входит в конвейер и следует по маршруту, на котором необходимым сервисам предоставляется экземпляр `DbContext`, создается одиночный экземпляр, используемый для всех сервисов, которые могут потребовать подключения к базе данных. По сути область действия сервиса, созданного контейнером, ограничивается запросом HTTP, и этот сервис используется для разрешения всех зависимостей во время запроса. При завершении запроса контейнер освобождает сервисы, чтобы они могли быть уничтожены исполнительной средой.

Ниже приведены некоторые примеры сервисов приложения, которые встречаются в проекте `AlpineSkiHouse.Web`. Жизненный цикл сервиса задается при помощи методов `Add*` с соответствующими именами.

```
services.AddSingleton<IAuthorizationHandler, EditSkiCardAuthorizationHandler>();
services.AddTransient<IEmailSender, AuthMessageSender>();
services.AddTransient<ISmsSender, AuthMessageSender>();
services.AddScoped<ICsrInformationService, CsrInformationService>();
```

С ростом списка сервисов может быть полезно создать методы расширения, упрощающие метод `ConfigureServices`. Например, если ваше приложение содержит

десятки классов `IAuthorizationHandler` для регистрации, возможно, вам стоит создать метод расширения `AddAuthorizationHandlers`.

Пример метода расширения для добавления набора сервисов

```
public static void AddAuthorizationHandlers(this IServiceCollection services)
{
    services.AddSingleton<IAuthorizationHandler,
        EditSkiCardAuthorizationHandler>();
    // Добавление других обработчиков
}
```

После добавления сервисов в `IServiceCollection` фреймворк берет на себя подключение зависимостей во время выполнения, используя внедрение через конструктор. Например, когда запрос перенаправляется `SkiCardController`, фреймворк создает новый экземпляр `SkiCardController`, передавая требуемые сервисы через открытый конструктор. Контроллер уже не знает, как создавать эти сервисы и как управлять их жизненным циклом.

ПРИМЕЧАНИЕ

При разработке новой функциональности иногда встречается ошибка, которая выглядит примерно так: «`InvalidOperationException: Не удастся разрешить сервис типа 'ServiceType' при попытке активизации 'SomeController'`» (рис. 14.1).

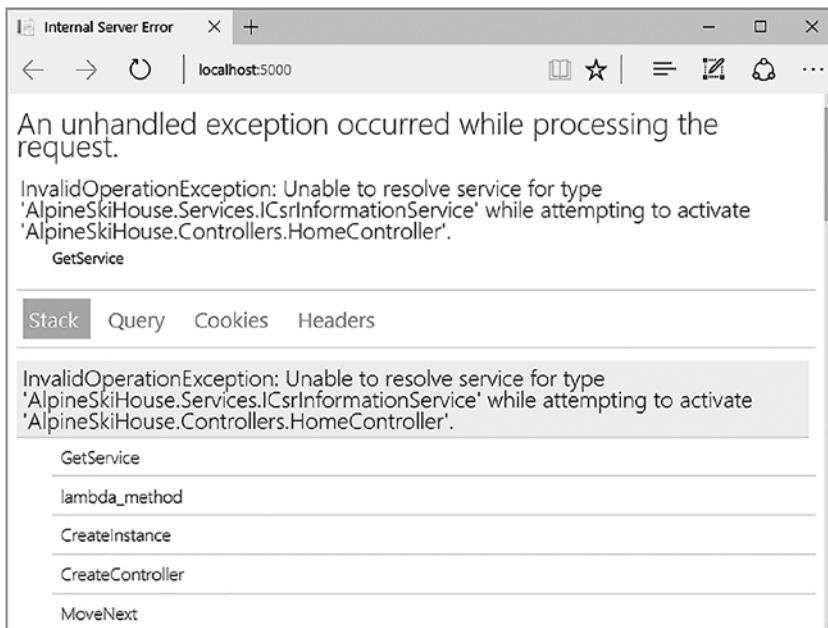


Рис. 14.1. Пример страницы ошибки, которая отображается в том случае, если необходимый сервис не был добавлен в коллекцию сервисов

Наиболее вероятная причина заключается в том, что вы забыли добавить тип сервиса в метод `ConfigureServices`. В этом случае проблема решается добавлением `CsrInformationService`.

```
services.AddScoped<ICsrInformationService, CsrInformationService>();
```

Использование стороннего контейнера

Встроенный контейнер, предоставляемый фреймворком ASP.NET Core, содержит только базовые функции, необходимые для поддержки большинства приложений. Тем не менее для .NET существует ряд других, более функциональных и проверенных временем фреймворков внедрения зависимостей. К счастью, архитектура ASP.NET Core такова, что контейнер по умолчанию может быть заменен сторонними контейнерами. Посмотрим, как это делается.

Среди популярных контейнеров IoC для .NET — Ninject, StructureMap и Autofac. На момент написания книги контейнер Autofac наиболее совершенен в своей поддержке ASP.NET Core, поэтому мы используем его в нашем примере.

Прежде всего, следует добавить ссылку на пакет NuGet `Autofac.Extensions.DependencyInjection`. Затем в метод `ConfigureServices` из `Startup.cs` вносятся некоторые изменения. Вместо `void` метод `ConfigureService` должен возвращать `IServiceProvider`. Сервисы фреймворка по-прежнему добавляются в `IServiceCollection`, а сервисы приложения регистрируются в контейнере Autofac. Наконец, возвращается объект `AutofacServiceProvider`, который передает ASP.NET Core контейнер Autofac вместо встроенного контейнера.

Сокращенная версия `ConfigureServices` с использованием `AutoFac`

```
public IServiceProvider ConfigureServices(IServiceCollection services)
{
    // Добавление сервисов фреймворка.
    services.AddDbContext<ApplicationUserContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddDbContext<SkiCardContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddDbContext<PassContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddDbContext<PassTypeContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddDbContext<ResortContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationUserContext>()
        .AddDefaultTokenProviders();
}
```

```

    services.AddOptions();

    services.AddMvc();

    // Теперь сервисы регистрируются в контейнере Autofac
    var builder = new ContainerBuilder();
    builder.RegisterType<CsrInformationService>().As<ICsrInformationService>();
    builder.Populate(services);
    var container = builder.Build();
    // Создание IServiceProvider на базе контейнера
    return new AutofacServiceProvider(container);
}

```

И хотя приведенный пример относительно тривиален, Autofac также предоставляет дополнительные расширенные возможности — например, сканирование сборок для поиска классов по установленному вами критерию. Сканирование сборок позволяет автоматически зарегистрировать все реализации `IAuthorizationHandler` в нашем проекте.

Использование сканирования сборки для автоматической регистрации типов

```

var currentAssembly = Assembly.GetEntryAssembly();

builder.RegisterAssemblyTypes(currentAssembly)
    .Where(t => t.IsAssignableTo<IAuthorizationHandler>())
    .As<IAuthorizationHandler>();

```

Другая интересная особенность Autofac — возможность разбиения конфигураций на модули. Модуль представляет собой обычный класс, который содержит конфигурацию взаимосвязанного набора сервисов. В простейшей форме модули Autofac похожи на модули расширения для `IServiceCollection`. Тем не менее модули также могут использоваться для реализации расширенной функциональности. Так как они являются классами, они также могут обнаруживаться на стадии выполнения и динамически загружаться для реализации инфраструктуры плагинов.

Простой пример модуля Autofac

```

public class AuthorizationHandlersModule : Module
{
    protected override void Load(ContainerBuilder builder)
    {
        var currentAssembly = Assembly.GetEntryAssembly();

        builder.RegisterAssemblyTypes(currentAssembly)
            .Where(t => t.IsAssignableTo<IAuthorizationHandler>())
            .As<IAuthorizationHandler>();
    }
}

```

Загрузка модуля в Startup.ConfigureServices

```

builder.RegisterModule(new AuthorizationHandlerModule());

```

Autofac — зрелый, полнофункциональный инструмент внедрения зависимостей, который здесь лишь кратко упоминается. За дополнительной информацией обращайтесь к превосходной документации Autofac по адресу <http://docs.autofac.org/>.

Итоги

Встроенная гибкая поддержка внедрения зависимостей в ASP.NET Core — желанное дополнение фреймворка. Использование внедрения зависимостей значительно упрощает создание экземпляров и управление сервисами с передачей обязанностей к контейнеру. Упрощается реализация контроллеров и сервисов вашего приложения, потому что им достаточно объявить сервисы, от которых они зависят. Как будет показано в главе 20 «Тестирование», при использовании внедрения через конструктор упрощается модульное тестирование.

Мы рекомендуем начать со встроенного контейнера, потому что предоставляемых им возможностей достаточно для большинства приложений. Если же вам потребуются какие-то нетривиальные возможности, вы в любой момент можете переключиться на сторонний контейнер. В следующей главе рассматривается роль JavaScript в современных веб-приложениях.

15

Роль JavaScript

Еще был только вторник, а у Даниэль уже болела голова. На прошлой неделе головная боль появилась только в четверг. Даниэль называла ее «синдромом JavaScript». Этот язык ей не нравился. Она считала, что это глупый неполноценный язык, и ее просто раздражало, что JavaScript не мешает ей совершать ошибки. Еще ее злил тот факт, что в JavaScript функции вызывают функции, генерирующие функции. Потерев виски, Даниэль решила, что пора выпить кофе.

По пути на кухню она прошла мимо офиса Честера. Там всегда отыскивалось что-нибудь интересное, и Даниэль остановилась, чтобы посмотреть, какие наклейки появились за последнюю неделю. Одна наклейка показалась знакомой: jQuery. Даниэль было знакомо это название; знакомо слишком хорошо. Оно было причиной ее головной боли в пятницу на прошлой неделе.

«У тебя новая наклейка с jQuery, Честер?»

«Ага, — ответил Честер. — Я продолжаю возиться с JavaScript. Смотри, я только что установил менеджер пакетов bower, используя npm, который работает в Node».

«Обалдеть! — Даниэль вскинула руки. — Не так быстро. Я еще не выпила кофе и не знаю, смогу ли переварить JavaScript без него».

«Хорошо, — сказал Честер. — Как насчет поработать на пару? В этой области есть много классных технологий, которыми мы можем пользоваться. Но это подождет, пока ты не выпьешь кофе».

«Даже не знаю, — ответила Даниэль. — Мне нужно влить в себя так много кофе, чтобы заглушить мою головную боль, а пить целый бассейн я буду очень-очень долго».

Программирование сайта в современном интернете объединяет программирование как на стороне сервера, так и на стороне клиента. На сервере выбор языков настолько широк, что он может сбить с толку. Даже если вы решили использовать язык .NET, все равно остаются варианты: F#, C# и даже VB.NET. Однако на стороне клиента вы ограничены кодом JavaScript.

В этой главе мы рассмотрим роль JavaScript в современной веб-разработке, обсудим, сколько именно нужно кода JavaScript, и поговорим о том, как настроить конвейер сборки для JavaScript. Затем будет рассмотрена компиляция в JavaScript и представлены некоторые фреймворки, упрощающие создание страниц. Напоследок вы узнаете, как организовать ваш код JavaScript, чтобы он не превращался в свалку.

Написание хорошего кода JavaScript

Язык JavaScript подталкивает разработчика к плохому стилю программирования. Возможно, кто-то сочтет это заявление спорным, но подумайте: в этом языке отсутствовали многие фундаментальные программные конструкции, такие как классы и пространства имен. Конечно, и классы, и пространства имен можно смоделировать в программе, но это потребует довольно высокой квалификации. Если обратиться к учебникам JavaScript начального уровня, ни в одном из них не рассказано о том, как создавать классы; с другой стороны, аналогичные учебники по C# просто вынуждены немедленно знакомить читателя с классами, потому что классы играют фундаментальную роль в языке. К счастью, в современном коде JavaScript некоторые недостатки были преодолены введением нового синтаксиса для классов и модулей. Эти конструкции используются и в этой главе, и в приложении Alpine Ski House в целом.

Метод включения JavaScript в страницу также создает риск применения плохого стиля. Ничто не мешает вам включать JavaScript прямо в HTML. Например, следующий код устанавливает розовый фон страницы после завершения загрузки:

```
<html>
<body onload="document.querySelector('body').style.backgroundColor = 'pink'">
```

Подобное смешение кода и разметки затрудняет сопровождение и отладку кода. Возможность выполнения кода JavaScript в странице также делает сайт уязвимым для атак XSS (cross-site scripting). В атаках этого типа атакующий помещает JavaScript в поля для ввода данных, предназначенные для других целей, а измененный сценарий отображается и выполняется последующими посетителями сайта. Представьте сайт, который может редактироваться любым желающим, — например, Википедию. Если бы Википедия не проверяла введенные данные, атакующий мог бы включить в страницу вызов `alert`, и все последующие посетители сайта получат сообщение:

```
alert("You are quite smelly");
```

Непрошеное сообщение — всего лишь досадная мелочь. В другой, гораздо более коварной атаке внедряется код JavaScript, который похищает содержимое поля паролей и отправляет информацию атакующему.

Атаки такого рода предотвращаются проверкой входных данных. Другое, еще более удачное решение основано на использовании заголовков **Content-Security Policy**. Эти заголовки, поддерживаемые большинством современных браузеров, запрещают браузеру выполнение любого встроенного кода JavaScript, фактически нейтрализуя угрозу атак XSS. Поддерживаться будет только код JavaScript, находящийся во внешнем файле сценариев.

ПРИМЕЧАНИЕ

Если вас интересует, какие браузеры поддерживают новые возможности, вам поможет сайт Can I Use по адресу <http://caniuse.com>. Например, матрица поддержки заголовков Content-Security Policy находится по адресу <http://caniuse.com/#feat=contentsecurity-policy>.

СОВЕТ

На канале ASP.NET Monsters записан эпизод, посвященный заголовкам Content-Security Policy (<https://channel9.msdn.com/Series/aspnetmonsters/ASP-NET-Monsters-66-Content-Security-Policy-Headers>). Просмотрите его для получения дополнительной информации.

Несомненно, следует использовать внешние файлы сценариев, включаемые в страницу тегом `script`. Другие преимущества хранения JavaScript в отдельных файлах — возможность прямой передачи сценариев ядру JavaScript для проведения модульного тестирования без необходимости запуска полноценного браузера. За дополнительной информацией о тестировании кода JavaScript и тестировании вообще обращайтесь к главе 20 «Тестирование».

А это вообще обязательно?

А ведь всего каких-то 22 года назад язык JavaScript еще не существовал. Так действительно ли необходимо использовать JavaScript на сайте? Честно говоря, да. JavaScript предоставляет всевозможную функциональность, к которой невозможно получить доступ другими способами. Расширенные пользовательские элементы управления, одностраничные приложения и даже метрики веб-сайтов — всеми этими аспектами управляет JavaScript. Без него все взаимодействия осуществлялись бы обратной отправкой на сервер, что создает задержку и требует более мощного сервера — а это, конечно, стоит дополнительных денег. Создание сайта без JavaScript отрицательно сказывается на функциональности.

Возможно, вы слышали о корректном снижении функциональности, чтобы пользователи с отключенной поддержкой JavaScript могли нормально пользоваться сайтом. Найти статистику о доле пользователей с отключенной поддержкой JavaScript за последние годы достаточно трудно. С другой стороны, доля таких

пользователей сокращается уже много лет, и сейчас разработка специальной версии сайта с перемещением всей функциональности на сервер уже не оправдана. Конечно, могут встречаться ситуации с высоким процентом пользователей, у которых отключение поддержки JavaScript оправданно, но такие случаи редки, и вы, скорее всего, будете знать, что оказались в такой ситуации.

Организация кода

В ASP.NET Core MVC появилась концепция каталога **wwwroot**, в котором хранятся все файлы контента, обрабатываемые веб-сервером. К числу таких файлов относятся таблицы стилей, графика и файлы JavaScript. В шаблоне по умолчанию эта папка содержит ряд ресурсов, существующих только в этом месте. А это означает, что при удалении папки **wwwroot** эти ресурсы будут потеряны. Таким образом, если вы используете шаблон по умолчанию, будет разумно сохранить папку **wwwroot** в системе управления кодом.

Впрочем, существует и более рациональный подход к управлению каталогом **wwwroot**. Все файлы в **wwwroot** следует хранить вне дерева, а содержимое **wwwroot** при необходимости должно воссоздаваться с использованием одних лишь средств сборки. Рассмотрим все типы файлов и место их хранения.

Файлы CSS управляют оформлением сайта. Тем не менее написание «чистого» кода CSS для большого сайта — сложный процесс. Исторически в CSS не было поддержки переменных, поэтому принцип DRY для него был не характерен. Для решения проблемы были созданы языки, компилируемые в CSS. Наиболее популярные инструменты такого рода — Less и SASS. Вместо того чтобы хранить файлы CSS в **wwwroot**, держите их вне корневого каталога в специальной папке (например, **Style**). В процессе сборки эти файлы преобразуются из SASS в CSS и размещаются в каталоге **wwwroot**.

Графические файлы также могут обрабатываться в процессе сборки. Графика в интернете — основная причина «разбухания» страниц. Чем больше изображение, тем дольше оно загружается и тем медленнее реагирует ваш сайт на действия пользователя. Часто используемые изображения можно оптимизировать и сохранить основное содержание, удалив бесполезные части. Для выполнения такой оптимизации существуют замечательные инструменты на базе Java, после применения которых изображения могут копироваться в **wwwroot**.

Наконец, файлы JavaScript почти всегда требуют последующей обработки. Они должны быть сокращены и, возможно, объединены для создания одного файла JavaScript. И снова файлы должны копироваться из другой папки в дерево. Мы рекомендуем начать с папки **Scripts**, находящейся вне **wwwroot**, и создать в ней дополнительные папки для разных типов сценариев. Конкретная структура папки **Scripts** зависит от сайта. Хорошие кандидаты — папки **Pages**, **Controls** и **Services** (рис. 15.1).

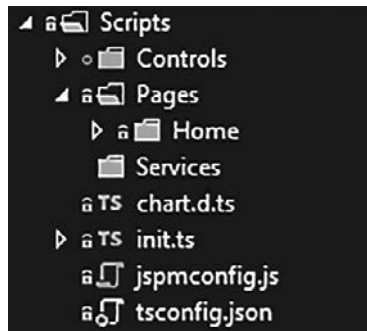


Рис. 15.1. Структура каталога Scripts

Короче говоря, к папке `wwwroot` следует относиться почти так же, как мы относимся к каталогу `bin` — как к временному месту хранения артефактов сборки.

SPA или не SPA?

Как узнать опытного разработчика? На технические вопросы он отвечает: «Это зависит от обстоятельств», прежде чем пускаться в долгие объяснения. На вопросы, касающиеся выбора технологий, очень редко можно дать простой ответ. Почти всегда приходится глубоко погружаться в пространство задачи, картина никогда не бывает черно-белой.

Этот принцип справедлив и для вопроса: «Сколько кода JavaScript следует использовать на сайте?» При этом приходится учитывать столько же факторов, сколько калорий в правильном филаделфийском сэндвиче с моцареллой и мясом. Разброс получается предельно широким, от использования JavaScript только для простых взаимодействий (например, проверки вводимых данных) до одностраничных приложений (SPA) и ситуаций, в которых все взаимодействия с пользователем происходят посредством JavaScript.

Приложения на стороне сервера — более традиционный метод построения веб-приложений. Логика приложения хранится на сервере, который отображает страницы в HTML и передает их браузеру. Эта модель включается в шаблон MVC по умолчанию. Каждая навигация переводит пользователя к новой странице, а взаимодействие с сервером обычно принимает форму получения новой страницы или отправки данных формы. В целом величина трафика между клиентом и сервером в приложениях на стороне сервера больше, чем в SPA, вследствие чего приложение работает медленнее. С другой стороны, трафик распределяется по всей продолжительности сеанса.

Одностраничные приложения изначально создают повышенную нагрузку на канал для передачи контента клиенту, но на протяжении всего пользовательского

сеанса они обычно оказываются более эффективными. Вы можете использовать сочетание шаблонов и данных с сервера, передаваемых в формате JSON, для создания разметки HTML на стороне клиента, работая с моделью DOM (Document Object Model).

Утверждение о том, что одностраничные приложения создавать труднее, чем приложения на стороне сервера, сейчас уже не столь однозначно. За последнее десятилетие фреймворки JavaScript на стороне клиента непрерывно совершенствовались. По эффективности разработки с использованием этих фреймворков одностраничные приложения уже практически не отстают от приложений на стороне сервера. Некоторые из этих фреймворков рассматриваются позже в этой главе.

Эффективность разработки также повышалась в результате совершенствования инструментов. Хотя JavaScript — единственный реальный вариант для программирования на стороне клиента, можно создавать языки, компилируемые в JavaScript. Такие языки, как CoffeeScript и TypeScript, делали это годами, и благодаря им появилась аналогия «JavaScript — это ассемблер для веб-программирования». Кстати, существует довольно интересный проект `asm.js` — подмножество JavaScript, спроектированное с расчетом на большую эффективность компиляции и выполнения. Также в `asm.js` могут компилироваться такие языки, как C++.

Существуют замечательные проверочные инструменты для поиска ошибок JavaScript, которые не препятствуют компиляции, но мешают нормальному выполнению. Запуск модульных тестов также может быть частью вашего конвейера сборки; кстати говоря, сам конвейер может создаваться с использованием мощных инструментов.

Сборка JavaScript

Выбор инструмента для сборки JavaScript из формата, используемого системой управления кодом, в форму, используемую клиентом, — задача исследования всего богатства выбора. Существуют инструменты, встроенные в Visual Studio, и инструменты, используемые большинством веб-разработчиков. Экосистема перемещается настолько быстро, что «лучшие» инструменты для конкретной ситуации постоянно меняются. Дело в том, что средства сборки не относятся к новым изобретениям. Программа Make существовала десятилетиями, и задача сборки особо не менялась. Исходные файлы должны пройти процесс преобразования из одного формата в другой. В программировании C файлы `.c` преобразуются в файлы `.o`, после чего компонуются в один двоичный файл (например, `a.out`). В программировании TypeScript мы берем файлы `.ts`, преобразуем их в файлы `.js`, после чего упаковываем в один сжатый файл. Аналогия очевидна, и даже современные инструменты не слишком далеко отходят от своих корней.

ПРИМЕЧАНИЕ

Мы уже начинаем постепенный переход с заслуженного протокола HTTP 1.1 на совершенно новый протокол HTTP 2.0. Этот протокол, основанный на протоколе Google SPDY, добавляет поддержку конвейерной передачи нескольких файлов по одному подключению. Это означает, что вам уже не придется сцеплять свои файлы в один файл, чтобы избежать непроизводительных затрат TCP/IP. Это первая серьезная модификация HTTP с 1999 года, которая вводит целый набор усовершенствований. За дополнительной информацией о HTTP 2.0 обращайтесь по адресу <https://tools.ietf.org/html/rfc7540>; также стоит ознакомиться с точкой зрения Пола-Хеннинга Кампа (Poul-Henning Kamp) по адресу <http://queue.acm.org/detail.cfm?id=2716278>.

Bundler & Minifier

Простейший инструмент обработки JavaScript — Bundler & Minifier — распространяется как расширение для Visual Studio. Компания Microsoft рекомендует использовать его в новых проектах ASP.NET Core. Bundler & Minifier жертвует частью мощности и гибкости ради простоты использования. В основном разработчику достаточно щелкнуть правой кнопкой мыши, выбрать команду Bundler & Minifier, после чего выбрать команду Minify File (рис. 15.2).

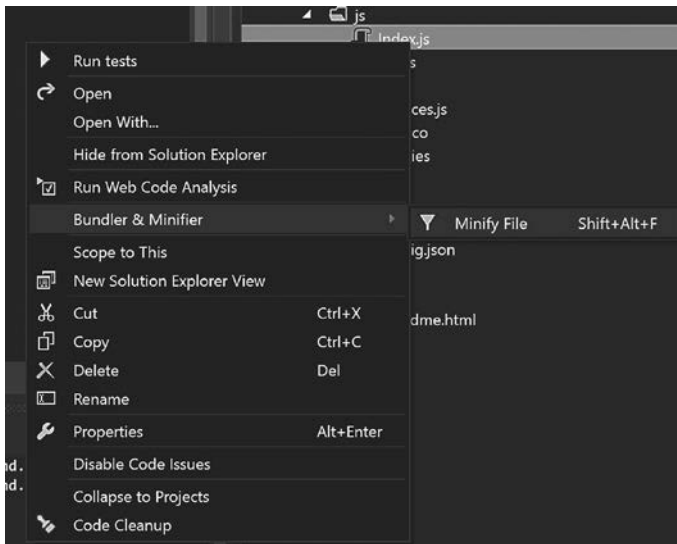


Рис. 15.2. Команда меню Bundler & Minifier

Если выделить несколько файлов, вы можете объединить их. Bundler & Minifier получает конфигурацию из файла `bundleconfig.json`, который фактически содер-

жит список соответствий между источниками и выходными файлами. Файл `bundleconfig.json` по умолчанию выглядит так:

```
[
  {
    "outputFileName": "wwwroot/css/site.min.css",
    "inputFiles": [
      "wwwroot/css/site.css"
    ]
  },
  {
    "outputFileName": "wwwroot/js/site.min.js",
    "inputFiles": [
      "wwwroot/js/site.js"
    ],
    "minify": {
      "enabled": true,
      "renameLocals": true
    }
  }
]
```

Одна из приятных особенностей Bundler & Minifier — возможность простого подключения к средствам командной строки .NET Core. Это позволит вам выполнять команды вида:

```
dotnet bundle
```

Команда читает файл `bundleconfig.json` и обрабатывает указанные в нем задачи конкатенации и сжатия для создания выходного файла. Bundler & Minifier также умеет отслеживать изменения в файлах, чтобы автоматически генерировать выходной файл при сохранении.

Мы рекомендуем использовать этот инструмент разве что в самых простейших случаях, потому что его возможности ограничены и он не является стандартным инструментом JavaScript. Сообщества Node, Go или Elixir не торопятся переходить на Bundler & Minifier, потому что существуют более совершенные, но не более сложные инструменты. Существует целый ряд возможностей, которые могут быть реализованы в полноценном инструменте построения JavaScript, но остаются недоступными для Bundler & Minifier.

Grunt

Grunt — первый инструмент построения JavaScript, который пользовался значительной популярностью. Он был построен на JavaScript и мог использовать другие библиотеки JavaScript для выполнения самых разнообразных действий. Также его представляли как «исполнитель задач», а это означает, что его возможности не

ограничиваются простой обработкой JavaScript. Существуют плагины для работы с CSS, контроля качества кода, модульного тестирования и уменьшения изображений. Количество плагинов приближается к 6000, так что экосистема у Grunt достаточно зрелая.

Чтобы начать работу с Grunt, необходимо установить его при помощи npm. Глобальная установка осуществляется следующей командой:

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

На самом деле эта команда устанавливает только инструментарий для поиска и запуска файла Gruntfile, поэтому вам еще придется установить Grunt в локальном проекте, для чего снова используется npm.

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

Команда добавляет зависимость Grunt в файл `package.json`. В нашем примере также будет установлен пакет `grunt-ts`. Этот пакет представляет собой задачу Grunt, которая может быть использована для построения файлов TypeScript. Мы рассмотрим TypeScript позже в этой главе, а пока поверьте — это замечательная программа.

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

После установки задачи Grunt можно создать очень простой файл `Gruntfile.js`, управляющий выполнением Grunt.

```
module.exports = function(grunt) {  
  grunt.initConfig({  
    ts: {  
      default: {  
        src: ["**/*.ts", "!node_modules/**"]  
      }  
    }  
  });  
  grunt.loadNpmTasks("grunt-ts");  
  grunt.registerTask("default", ["ts"]);  
};
```

Здесь экспортируется функция, которая получает объект `grunt` для обработки. Модуль импортируется в Grunt, обрабатывается и используется напрямую. Так как он содержит чистый код JavaScript, в него можно включить любой синтаксически правильный код JavaScript. В нашем случае работа начинается с настройки конфигурации для TypeScript и задачи `ts`.

Свойство `src` используется для определения маски файлов, используемых при компиляции. Все содержимое каталога `node_modules` игнорируется, а все файлы с расширением `.ts` включаются в обработку.

Затем загружается библиотека `grunt-ts`, предоставляющая непосредственную функциональность компиляции TypeScript в JavaScript. В завершение регистрируется задача `default`, чтобы у нее существовала зависимость от задачи `ts`. Теперь можно выполнить в командной строке следующую команду:

```
grunt
Running "ts:default" (ts) task
Compiling...
Cleared fast compile cache for target: default
### Fast Compile >>Scripts/chart.d.ts
### Fast Compile >>Scripts/Controls/MetersSkied.ts
### Fast Compile >>Scripts/Controls/MetersSkiedLoader.ts
### Fast Compile >>Scripts/init.ts
### Fast Compile >>Scripts/Pages/Home/Blah.ts
### Fast Compile >>Scripts/Pages/Home/Index.ts
Using tsc v1.8.10
```

TypeScript compilation complete: 1.86s for 6 TypeScript files.

Done.

Как видно из вывода, ряд файлов компилируется из TypeScript в JavaScript. Последующий шаг построения может объединить их в один файл и переместить в `wwwroot`.

gulp

Одна из претензий к Grunt заключается в том, что программа оставляет большое количество промежуточных файлов. Обработка JavaScript может состоять из нескольких шагов, и после каждого шага Grunt оставляет файлы, которые приходится удалять позже.

Gulp решает эту проблему. Принцип работы gulp основан на концепции потоков. Каждому шагу процесса сборки ставится в соответствие поток, который может направляться через серию каналов (рис. 15.3).

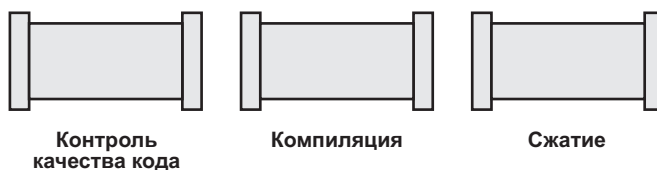


Рис. 15.3. Конвейер сборки

Вывод каждого этапа в канале становится вводом для следующего этапа, поэтому временные файлы на диск не записываются.

Модель потоков превосходно подходит для процесса построения в целом, а метафора потока достаточно часто используется в пространстве задач. Программа `gulp`, как и `Grunt`, имеет очень богатую экосистему. Для нее написаны тысячи плагинов (свыше 2500 на момент написания книги). Для управления `gulp` также используется простой файл JavaScript, поэтому он может содержать любой синтаксически правильный код JavaScript и использовать многочисленные модули, содержащиеся в `npm`.

Начнем с рассмотрения базового файла `gulp`, а затем доработаем его для выполнения всех процессов сборки.

```
var gulp = require("gulp"),
    rename = require("gulp-rename"),
    typescript = require("gulp-typescript");

var webroot = "./wwwroot/";
var sourceroot = "./Scripts/";

var paths = {
  ts: sourceroot + "**/*.ts",
  tsDefinitionFiles: "npm_modules/@types/**/*.d.ts",
  jsDest: webroot + "js/"
};

gulp.task("typescript", function(){
  return gulp.src([paths.tsDefinitionFiles, paths.ts, "!" + paths.minJs],
    { base: "." })
    .pipe(typescript({
      module: "system"
    }))
    .pipe(rename((pathObj, file) => {
      return pathObj.join(
        pathObj.dirname(file).replace(/^Scripts\/?\?\//, ''),
        pathObj.basename(file));
    }))
    .pipe(gulp.dest(paths.jsDest));
});

gulp.task("default", ["typescript"]);
```

В первых трех строках файла подгружаются модули: собственно `gulp`, библиотека `rename` и задача `typescript`. Это простые модули `npm`, для добавления которых используется стандартный синтаксис `require`. Затем в файле создаются переменные, чтобы нам не приходилось использовать «волшебные строки» в оставшейся

части файла. Обратите внимание на синтаксис маски ****** — он обозначает все каталоги любой степени вложенности, что позволяет нам хранить файлы TypeScript в хорошо организованной структуре папок.

В оставшейся части файла определяется пара задач. Первая задача явно интереснее: сначала директива **gulp.src** собирает все исходные файлы. Затем эти файлы передаются задаче компилятора **typescript**. Мы подготовили настройки для генерации модулей в стиле системных файлов, их мы рассмотрим позже. Затем файлы обрабатываются задачей **rename**, которая отсекает префикс **Scripts**. Того же результата можно было добиться изменением базового каталога для компиляции TypeScript, но тогда мы бы лишились столь замечательной демонстрации каналов. Наконец, файлы передаются в каталог-приемник.

Подгрузка дополнительных библиотек позволяет выполнить такие действия, как оптимизация графики. Для **gulp** существует замечательный плагин **gulp-image-optimization**, которым можно воспользоваться для этой цели. Для этого достаточно включить следующую библиотеку:

```
var imageop = require('gulp-image-optimization');
```

Затем напишите задачу для выполнения оптимизации:

```
gulp.task("images", function()
{
    return gulp.src(paths.images)
        .pipe(imageop({
            optimizationLevel: 5,
            progressive: true,
            interlaced: true
        })).pipe(gulp.dest(paths.imagesDest));
});
```

Обратите внимание: этот код имеет такую же базовую структуру, как и компиляция TypeScript. Он задает поток при помощи **src** и передает вывод по цепочке из нескольких этапов. Полный пример файла **gulp** для Alpine Ski House содержится в репозитории GitHub. Ваши возможности при использовании **gulp** практически не ограничены; это потрясающий инструмент.

Пожалуй, **gulp** на данный момент является самым популярным инструментом построения JavaScript. Впрочем, WebPack тоже пользуется вниманием разработчиков.

WebPack

WebPack считается решением для больших проектов, которые должны динамически загружать фрагменты JavaScript. Проблема с большими проектами заключа-

ется в том, что вам приходится либо нести затраты заранее и загружать огромный блок JavaScript при первоначальной загрузке, либо жонглировать с загрузкой модулей с сервера по мере надобности, что замедляет работу приложения.

ПРИМЕЧАНИЕ

Если вы захотите больше узнать о различных способах передачи файлов JavaScript с сервера на сторону клиента, на канале ASP.NET Monsters был записан эпизод как раз по этой теме. Ролик находится по адресу <https://channel9.msdn.com/Series/aspnetmonsters/ASPNET-Monsters-Ep-67-Getting-JavaScript-to-the-Client>.

Попробуем применить WebPack в примере с компиляцией TypeScript, который уже использовался ранее. Но перед этим необходимо сказать пару слов о типах модулей JavaScript. Существуют по крайней мере пять основных стандартов определения модулей в JavaScript (CommonJS, AMD, System, ES2015 и UMD) и еще пара десятков менее известных форматов. Проблема в том, что в JavaScript формат модулей только недавно был определен как часть языка. Природа не терпит пустоты, поэтому вполне естественно, что появился целый ворох модулей, которые мало чем отличались друг от друга. Компилятор TypeScript может генерировать любой из пяти основных форматов в зависимости от значения переданного флага `module`. WebPack поддерживает модули CommonJS и AMD, но не поддерживает никакие другие.

Следовательно, необходимо позаботиться о том, чтобы компилятор TypeScript генерировал модули CommonJS. Для этого можно поместить в корневой каталог проекта файл `tsconfig.json` с настройками TypeScript в проекте. Простейший файл может выглядеть так:

```
{
  "compilerOptions": {
    "module": "commonjs",
    "noImplicitAny": true,
    "removeComments": true,
    "preserveConstEnums": true
  }
}
```

В этом фрагменте самая важная строка — первая, определяющая тип создаваемого модуля. Конечно, есть много других настроек, которые можно добавить в этот файл, но они здесь не рассматриваются. Когда файл будет готов, можно переходить к добавлению WebPack.

Сначала необходимо установить инструментарий WebPack глобально:

```
npm install -g webpack
```

В отличие от Grunt, эта команда устанавливает WebPack полностью, и вам не придется локально устанавливать дополнительные средства для работы WebPack. Тем не менее следует установить плагин для компиляции TypeScript:

```
npm install ts-loader --save
```

После этого создайте для WebPack файл сборки с именем `webpack.config.js`:

```
module.exports = {
  entry: './Scripts/Init.ts',
  output: {
    filename: 'wwwroot/js/site.js'
  },
  resolve: {
    extensions: ['', '.webpack.js', '.web.js', '.ts', '.js']
  },
  module: {
    loaders: [
      { test: /\.ts$/, loader: 'ts-loader' }
    ]
  }
}
```

Этот файл интересен не только тем, что он содержит, но и тем, чего в нем нет. Он содержит намного меньше информации, чем эквивалентный файл `gulp`. Файл WebPack определяет несколько правил на основании расширений файлов. Например, секция `module` определяет, какой инструмент должен использоваться для файла, имя которого совпадает с регулярным выражением `\.ts$`: это любой файл, завершающийся расширением `.ts`. Все такие файлы передаются плагину TypeScript. Следующее, на что следует обратить внимание, — определение точки входа в приложение. От этой точки WebPack анализирует включенные модули, чтобы определить, какие файлы должны быть включены в вывод. Например, мы передаем точку входа `init.ts`, которая включает файл `home/index.ts`, а тот в свою очередь включает `home/blah.ts`. Таким образом WebPack может определить файлы, включаемые в пакет, и включить `init.ts`, `home/index.ts` и `home/blah.ts`, игнорируя ряд других файлов `.ts` в разных местах проекта. Это означает, что если вы будете действовать последовательно при включении модулей командой `import`, WebPack сможет исключить лишний код из рабочего файла JavaScript.

Предположим, наряду с добавлением TypeScript и конкатенацией в один файл также необходимо сжать JavaScript. Задача решается при помощи плагина `UglifyJsPlugin`. Для этого необходимо установить WebPack в локальный каталог `node_modules`:

```
Npm install webpack
```

Затем этот плагин необходимо подключить в файле `webpack.config.js` и добавить в секцию `plugins`:

```
var webpack = require('webpack');
module.exports = {
  entry: './Scripts/Init.ts',
  output: {
    filename: 'wwwroot/js/site.js'
  },
  resolve: {
    extensions: ['', '.webpack.js', '.web.js', '.ts', '.js']
  },
  plugins: [
    new webpack.optimize.UglifyJsPlugin()
  ],
  module: {
    loaders: [
      { test: /\.ts$/, loader: 'ts-loader' }
    ]
  }
}
```

Вот и все, что необходимо для включения сжатия JavaScript в проект!

WebPack — мощный инструмент, но его внутренние механизмы скрыты от разработчика. Подобные инструменты хорошо работают, если следовать канонам, но стоит вам свернуть с проторенной дороги — они превращаются в кандалы. К достоинствам WebPack можно отнести возможность подключения `gulp` (а точнее, любого «чистого» кода JavaScript) в процесс сборки. Другая крайне интересная особенность WebPack — поддержка «горячей перезагрузки» JavaScript на активной веб-странице. Она упрощает разработку сложных одностраничных приложений, потому что вам уже не нужно полностью перезагружать страницу при каждом изменении JavaScript.

Какой инструмент мне лучше подойдет?

Невозможно сказать, какой инструмент лучше подойдет для вашей команды, потому что у каждого инструмента есть свои достоинства и недостатки. Команда Alpine Ski House использовала в своей работе `gulp`. В WebPack при всех его достоинствах многие моменты неочевидны, а потоковый подход казался более естественным, чем подход Grunt.

Может оказаться, что к моменту выхода этой книги все эти инструменты выйдут из числа фаворитов JavaScript и будут заменены чем-то совершенно новым. Раз-

работка JavaScript сейчас развивается невероятными темпами, что не обязательно является хорошим признаком. Не увлекайтесь новыми «модными» технологиями. Проверенные временем инструменты — такие, как Grunt и gulp, — также заслуживают вашего внимания.

TypeScript

Мы уже несколько раз упоминали TypeScript в этой главе, не рассказывая подробно о том, что же это такое. Пора прояснить ситуацию. TypeScript — язык, компилируемый в JavaScript. В этом отношении он похож на более известный язык CoffeeScript, но в отличие от последнего, TypeScript является надмножеством JavaScript. Из этого следует, что любой действительный код JavaScript автоматически является синтаксически правильным кодом TypeScript.

Впрочем, язык TypeScript — нечто гораздо большее, чем простое подмножество JavaScript, потому что в нем добавлена очень полезная функциональность. На наш взгляд, в TypeScript сочетаются сразу два инструмента: полнофункциональный компилятор ES2015 в ES5 и статическая система проверки типов с возможностью вывода типов.

Компилятор ES2015 в ES5

ECMAScript (или ES) — стандарт языка, известного как JavaScript. У JavaScript чрезвычайно динамичное сообщество, которое в течение многих лет сдерживалось крайне медленным темпом развития самого JavaScript. За последние несколько лет разработка ускориалась. Стандарт ES2015 был выпущен в 2015 году; в нем было определено огромное количество усовершенствований JavaScript. Было решено, что каждый год будет выпускаться новая версия JavaScript, в имя которой будет включаться год ее утверждения. В самом деле, существует ES2016, и известны планы на ES2017. ES2016 — относительно второстепенная версия, поэтому обычно мы продолжаем называть ее ES2015 или ES6 (хотя ко второму варианту многие относятся неодобрительно).

К сожалению, не все браузеры поддерживают новый стандарт ES2015. Если вы занимаетесь веб-разработкой в течение какого-либо времени, возможно, вы сейчас качаете головой и обвиняете Internet Explorer в том, что он тормозит развитие веб-технологий. Впрочем, на этот раз возлагать вину исключительно на Microsoft было бы неверно. Компания прикладывала огромные усилия, чтобы пользователи обновлялись до новейшей версии Internet Explorer и даже переходили на Edge — более современную реализацию браузера. Главными виновниками в наши дни на самом деле являются мобильные устройства, производители которых ограничивают пользователя определенной версией браузера. Обновление происходит очень медленно, и этот факт нужно учитывать. Именно здесь в игру вступают компиля-

торы JavaScript (обычно их называют «транскомпиляторами» или «транспиляторами»).

Существует целый ряд превосходных транскомпиляторов для преобразования ES2015 JavaScript в более распространенную разновидность ES5. В них реализованы разные уровни поддержки ES2015+, но TypeScript входит в число лидеров. TypeScript может взять класс следующего вида:

```
import {MembershipGraph} from "./MembershipGraph";

export class Index {
  constructor() {
    var graph = new MembershipGraph();
    console.log("Graph loaded.");
    window.setTimeout(()=> console.log("load complete"), 2000);
  }
}
```

и преобразовать его в эквивалент ES5, который выглядит так:

```
System.register(["./MembershipGraph"], function(exports_1, context_1) {
  "use strict";
  var __moduleName = context_1.id;
  var MembershipGraph_1;
  var Index;
  return {
    setters:[
      function (MembershipGraph_1_1) {
        MembershipGraph_1 = MembershipGraph_1_1;
      },
    ],
    execute: function() {
      Index = (function () {
        function Index() {
          var graph = new MembershipGraph_1.MembershipGraph();
          console.log("Graph loaded.");
          window.setTimeout(function () { return console.log
            ("load complete"); },
            2000);
        }
        return Index;
      })();
      exports_1("Index", Index);
    }
  });
});
```

Как видите, откомпилированная версия заметно больше исходной. Какую из них вы бы предпочли сопровождать? Обратите внимание на лямбда-синтаксис, который был расширен в эквивалентную функцию. Кроме расширения лямбда-

синтаксиса, также обратите внимание на значительный объем сгенерированного кода, относящегося к регистрации модуля в `System.register`. Дело в том, что мы попросили TypeScript сгенерировать модули в стиле SystemJS. Импортирование `MembershipGraph` было расширено в требование загрузки модуля `Index`. Вскоре мы расскажем о загрузчиках модулей более подробно, но генерирование такого шаблонного кода — одна из самых сильных сторон TypeScript.

ПРИМЕЧАНИЕ

Если вы захотите узнать, в каких транскомпиляторах и браузерах реализована лучшая поддержка новых возможностей JavaScript, загляните на сайт <https://kangax.github.io/compat-table/es6/>. Здесь вы найдете исключительно актуальную сравнительную таблицу.

В ES2015 слишком много замечательных возможностей, чтобы их можно было перечислить и объяснить в одной главе книги. По этой теме написаны целые книги. Единственное, о чем мы не могли не упомянуть, — это оператор области видимости переменной `let`. Видимость переменных в JavaScript всегда была немногим странной. В большинстве языков программирования используется видимость уровня блоков, но в JavaScript при использовании ключевого слова `var` используется видимость уровня функций. И хотя эта особенность открывает ряд интересных возможностей, в частности при работе с лямбда-конструкциями, она может стать причиной недоразумений. Любая переменная, объявленная с ключевым словом `let`, имеет блочную область видимости, намного более понятную для разработчиков с опытом работы на почти любом другом языке.

Даже если бы компиляция ES2015+ в ES5 была единственной особенностью TypeScript, программа была бы сильным конкурентом для таких компиляторов, как Babel; тем не менее компиляция — всего лишь половина того, что умеет TypeScript.

Система типизации

Среди многочисленных характеристик, используемых для классификации языков, встречается система типизации. Языки с сильной типизацией выполняют проверку типов на стадии компиляции; иначе говоря, если вы попытаетесь выполнить строковую операцию (скажем, замену символов) с целым числом, компилятор вас остановит. Примеры языков с сильной типизацией — F#, Java и C#. На другом конце оси находятся языки с динамической типизацией, у которых такие проверки выполняются только во время выполнения. Динамические языки вообще не интересуются тем, как выглядит объект; они концентрируются на одном конкретном методе, который вы вызываете. JavaScript принадлежит к числу языков с динамической типизацией, как и Ruby, Python и PHP.

Большие и сложные приложения вполне можно строить на языках с динамической типизацией. Тем не менее было бы полезно получать предупреждения о возможных проблемах на стадии компиляции или редактирования. Как правило, проблемы в это время решаются проще и дешевле, чем на стадии выполнения,

и чем раньше будут выявляться ошибки, тем лучше. TypeScript добавляет к традиционному JavaScript систему типизации, причем делает это с минимальными хлопотами (а то и вовсе без них).

Рассмотрим функцию для умножения двух чисел:

```
function multiply(one, two){
  return one * two;
}
```

Если вы используете вызов `multiply(2, "elephant")`, никаких проблем не будет до стадии выполнения; в этот момент вы получите ошибку NaN. Но если добавить аннотацию, которая указывает, что `one` и `two` — целые числа, TypeScript сможет сообщить об ошибке:

```
function multiply(one: number, two: number){
  return one * two;
}
```

Во время компиляции будет получено сообщение об ошибке «Аргумент типа 'string' не может быть присвоен параметру типа 'number'». Пример может показаться небольшим и тривиальным, но при построении все более сложных приложений полезность системы типов быстро становится очевидной. Не каждой переменной обязательно должен назначаться тип. Для некоторых переменных тип может быть определен компилятором; в остальных случаях TypeScript считает, что переменная имеет тип `any`, то есть никакие проверки для нее не выполняются.

Если вы используете библиотеку (например, `d3.js`), которая изначально не была написана на JavaScript, вам может пригодиться файл определений. Файл определений напоминает заголовочный файл C: он тоже определяет, какие функции содержит библиотека и какие входные и выходные типы в ней используются. Почти для каждой популярной библиотеки JavaScript доступны файлы определений, которые гарантируют, что вы правильно взаимодействуете с библиотекой. Например, команда Alpine Ski House включила зависимость от `chart.js` и создала файл определений, по которому проверяется правильность вызова функций построения графиков. Фрагмент файла определений выглядит так:

```
interface Chart {
  Line(data: LinearChartData, options?: LineChartOptions): LinearInstance;
  Bar(data: LinearChartData, options?: BarChartOptions): LinearInstance;
  Radar(data: LinearChartData, options?: RadarChartOptions): LinearInstance;

  PolarArea(data: CircularChartData[], options?: PolarAreaChartOptions):
    CircularInstance;
  Pie(data: CircularChartData[], options?: PieChartOptions): CircularInstance;
  Doughnut(data: CircularChartData[], options?: PieChartOptions):
    CircularInstance;
}
```

Как видите, в этом определении интерфейса `Chart` каждому параметру назначен тип. Наличие типов гарантирует, что все необходимые параметры будут заданы и им будут присвоены соответствующие значения.

Кроме того, использование TypeScript повышает качество автозаполнения в редакторах. В примере из редактора кода Visual Studio на рис. 15.4 показаны не только функции, предоставляемые объектом `chart`, но и подробные сведения о параметрах.

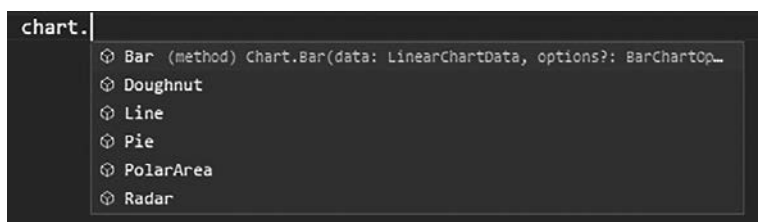


Рис. 15.4. Раскрывающийся список с вариантами автозаполнения

СТОИМОСТЬ ВЫХОДА

Принимая решение об использовании новой технологии, важно учитывать затраты на отказ от технологии, если ваша команда решит, что это решение не подходит. Одна из замечательных особенностей JavaScript заключается в том, что стоимость выхода практически равна нулю: достаточно перекомпилировать все файлы `.ts` в `.js` и удалить файлы `.ts`. Сгенерированный код нормально читается, у вас даже остается возможность сохранить комментарии. Кроме того, стоимость реализации TypeScript очень низка. Так как весь код JavaScript является синтаксически правильным кодом TypeScript, преобразование многих файлов сводится к простой замене `js` на `ts`.

Обратите внимание: мы говорим о «синтаксически правильном» коде TypeScript, а не просто о «коде TypeScript». Дело в том, что компилятор TypeScript выполняет проверку типов для любых фрагментов вашего кода JavaScript, которые будут для него доступны. Компилятор даже может найти в вашем коде JavaScript ошибки, о существовании которых вы и не подозревали!

Чтобы установить TypeScript, достаточно ввести команду:

```
npm update -g typescript
```

Конечно, если вы воспользовались инструментами, упомянутыми в предыдущем разделе, вы уже знаете, как компилировать TypeScript. TypeScript — замечательное дополнение JavaScript и мощный инструмент для управления любой большой кодовой базой JavaScript.

Загрузка модуля

В предыдущем разделе мы упоминали о том, что компилятор TypeScript для приложения Alpine Ski House настроен на создание модулей в стиле system.js. Это объясняется тем, что команда остановилась на использовании загрузчика модулей system.js для подключения файлов JavaScript. Загрузчики модулей добавляют возможность объединять отдельные файлы JavaScript по мере необходимости. Модули в основном могут просто рассматриваться как JavaScript-эквиваленты классов VB или C#.

System.js — библиотека, поддерживающая загрузку модулей из отдельных файлов на диске или из одного сцепленного файла. Загрузчик следит за тем, какие модули были загружены с сервера, чтобы повторное импортирование модулей не приводило к созданию лишнего сетевого запроса. На начальном этапе работа с system.js сводится к подключению библиотеки и ее использованию для загрузки файлов.

```
<script src="/js/system.js"></script>
<script src="/js/jspmconfig.js"></script>
<script> System.import("Pages/Home/Index");</script>
```

В этом примере вызов `System.import` запрашивает и загружает `/js/Pages/Home/Index`. Префикс `/js` предоставляется файлом `jspmconfig.js`. Этот файл является второй частью загрузки модулей в system.js.

jspm — менеджер пакетов для библиотек JavaScript. Во многих отношениях он идентичен Bower, но библиотеки в jspm обычно контролируются более тщательно, чем библиотеки Bower, для работы которых часто приходится дополнительно возиться с настройками. Jspm также может загружать библиотеки прямо с GitHub или npm; это гарантирует, что в вашем распоряжении всегда будут самые последние версии библиотек. Кроме того, jspm использует стандартные форматы модулей, а для Bower характерна невообразимая мешанина разных форматов.

Чтобы начать работу с jspm, необходимо сначала установить библиотеку из npm:

```
npm install jspm -g
```

После глобальной установки jspm выполняются почти такие же действия, как и при настройке npm для проекта.

```
jspm init
```

Команда задает ряд вопросов для построения файла `jspmconfig.js`, в котором среди прочего указан префикс `/js`, упоминавшийся ранее. Установка пакетов теперь сводится к выполнению простой команды:

```
jspm install chartjs
```

Результат выглядит так:

```
Updating registry cache...
Looking up github:chartjs/Chart.js
ok Installed chart as github:chartjs/Chart.js@^2.2.2 (2.2.2)
ok Install tree has no forks.

ok Install complete.
```

Менеджер jspm нашел библиотеку и установил ее с GitHub в формате, который может быть загружен одним из двух способов:

```
<script>System.import("chart").then( function(chart){
//do things with chart
});</script>
```

или из файла TypeScript:

```
import {Chart} from "chart";
```

При помощи JSPM (или любого другого инструмента) вы можете сцепить и упаковать ресурсы JavaScript для загрузчика в один файл. Впрочем, если и клиенты, и серверы поддерживают HTTP 2, можно просто разрешить system.js загружать файлы по мере необходимости.

System.js и jspm проектировались с расчетом на совместную работу, но, конечно, в этой области есть и другие инструменты. Например, чрезвычайно популярен загрузчик require.js. Команда Alpine Ski House выбрала комбинацию System.js и jspm из-за отличной интеграции и хорошей поддержки.

Выбор фреймворка

Если у вас голова идет кругом от количества типов модулей, менеджеров пакетов или инструментов сборки в сообществе JavaScript, перед чтением этого раздела вам лучше сесть. Существуют буквально тысячи фреймворков JavaScript, включая Angular, React, Aurelia, Backbone, Meteor и Ember.js. Выбрать, какой из них лучше подойдет для вашего проекта, будет весьма непросто. Трудно даже перечислить факторы, которые следует учитывать при выборе. Например, можно ориентироваться на распространенность, наличие квалифицированной поддержки, стабильность, возраст, функциональность, скорость, удобство сопровождения и кроссбраузерную поддержку. Также возможно пойти по пути, по которому идет большинство стартапов, и выбрать фреймворк, чаще всего упоминаемый в подфоруме веб-разработки на Reddit.

Для приложения Alpine Ski House команда решила использовать библиотеку представлений React в сочетании с библиотекой однонаправленной передачи ReluxJS.

React — исключительно популярная библиотека, которая поддерживается Facebook и интенсивно используется на сайте. Она использует виртуальную модель DOM для быстрого изменения DOM без лишней передачи данных или полной перерисовки частей страницы. React — компонентно-ориентированный фреймворк, предназначенный для создания нестандартных тегов. Это минималистичная библиотека (в отличие от таких продуктов, как AngularJS), поэтому ее приходится использовать в сочетании с другими инструментами — такими, как ReluxJS.

React хорошо сочетается с TypeScript, потому что новые версии TypeScript поддерживают файлы `.tsx`. «Что такое файл `.tsx`?» — спросите вы. Это просто TypeScript-версия файла `.jsx`. «Допустим, а что такое файл `.jsx`?» JSX — довольно странная смесь JavaScript и XML, чрезвычайно популярная в сообществе React. Например, поле для ввода числа в React может определяться следующим образом:

```
function() {  
  return (<div>  
    <label for={this.state.controlName}>{this.state.label}</label>  
    <input type="number" name={this.state.controlName}/>  
  </div>);  
}
```

На первый взгляд код выглядит очень странно, но если вы к нему привыкнете, React и JSX становятся чрезвычайно эффективными инструментами. TypeScript понимает этот синтаксис и обеспечивает проверку типов для свойств, встроженных в HTML.

Команда Alpine Ski House с самого начала решила, что она не будет использовать jQuery. Многим такое решение кажется чуть ли не святотатством. Строить веб-сайт без jQuery? Немыслимо. И правда, библиотека jQuery заполняла очень большой пробел в функциональности в старых браузерах. Однако с того времени язык JavaScript развивался, и в нем появились свои, намного более эффективные версии многих функциональных аспектов (например, `querySelector`).

ПРИМЕЧАНИЕ

Не хотите использовать jQuery? Тогда сайт <http://youmightnotneedjquery.com/> вам безусловно пригодится. На сайте не только собраны эквиваленты многих популярных функций jQuery на «чистом» JavaScript, но и приведены рекомендации по выбору более компактных и специализированных библиотек, которые могут использоваться вместо jQuery.

Итоги

Как ни прискорбно, объем этой главы (да в общем-то и всей книги) не позволяет в полной мере исследовать все уголки экосистемы JavaScript. Возможно, еще более печально то, что ни одну книгу по этой теме не удастся написать настолько

быстро, чтобы она не устарела в тот момент, когда высохнет типографская краска. JavaScript развивается быстрее, чем любой из известных нам языков программирования или экосистем. При этом он является неотъемлемой частью любого проекта веб-разработки. Возможно, лучший совет по работе с JavaScript встречается в трудах Дж. Р. Толкина: «В истинном золоте блеска нет»¹.

Инструменты JavaScript, лучше всего подходящие для вашего проекта, не всегда выглядят блистательными воплощениями крутизны JavaScript; это вполне могут быть надежные, хорошо поддерживаемые библиотеки и инструменты. Что бы вы ни выбрали, всегда приятно знать, что создатели ASP.NET Core уделяют самое пристальное внимание передовым методам работы с JavaScript. Осталась в прошлом компиляция JavaScript на стадии выполнения или такие смехотворные абстракции, как `UpdatePanel`. JavaScript наконец-то занимает место полноправного языка. В следующей главе рассматривается механизм внедрения зависимостей и преимущества его использования.

¹ В переводе И. Гриншпуна. — *Примеч. пер.*

16

Управление зависимостями

Разработка набирала темп. Новые версии передавались на тестирование до 20 раз в день. В списке ошибок обычно было не более двух-трех позиций — группа разработки расправлялась с проблемами в том же темпе, в котором они появлялись. Даниэль была в восторге от того, как много нового она узнавала. Она возвращалась домой поздно вечером, читала блоги и смотрела видеоролики. Энтузиазм таких людей, как Марк-1, был заразителен. И такое настроение было не только у нее. Не проходило и дня, чтобы кто-нибудь из участников не обнаруживал новую библиотеку, инструмент или концепцию, которая должна была улучшить их приложение.

«А ты видел библиотеку GenFu? — спросил Арджун однажды. — Она генерирует реалистичные тестовые данные на основе простого анализа имен классов. Смотри!» Арджун обновил страницу в браузере, и на экране появился список лыжников. «Видишь, поле имени автоматически заполняется: Лионель Пенушо. Возможно, это даже имя настоящего человека. Нашей группе контроля качества это понравится. Пожалуй, я добавлю эту библиотеку в репозиторий!»

Даниэль на секунду задумалась. Ведь она на прошлой неделе читала что-то о добавлении зависимостей в репозиторий? «Погоди, Арджун, я не думаю, что нам стоит добавлять этот файл. Думаю, его можно просто включить в список зависимостей».

«О нет! — воскликнул Арджун. — Я однажды так поступил в одном проекте. У всех были разные версии одной библиотеки, ничего не работало, потом всех уволили, и мне пришлось жить в машине... Ладно, с последним я перегнул, но идея все равно ужасная».

«Нет, нет, послушай меня. Я только вчера прочитала об одной программе — то ли `pougat`, то ли `pugget`, или что-то в этом роде. Она может управлять всеми зависимостями, и даже зависимостями зависимостей, используя транзитные замыкания».

«Транзитные замыкания? Ты хочешь сказать — транзитивные замыкания? — спросил Арджун. — Я и не знал, что такие программы существуют. Было бы здорово использовать нечто такое. Как ты думаешь, такие инструменты есть и для JavaScript?»

«Готова поспорить, что есть. Давай поищем».

В наши дни приложения более чем когда-либо строятся из множества мелких компонентов. Часто такими компонентами оказываются библиотеки, написанные другими разработчиками и используемые в тысячах других приложений. В прошлом разработчики часто загружали библиотеки и фиксировали их в системах управления кодом вместе с исходным кодом самого приложения. Процесс быстро стал утомительным, особенно когда библиотеки начали зависеть от конкретных версий других библиотек. Много проблем создавал переход на обновленную версию конкретной библиотеки, потому что можно было случайно нарушить работу другой библиотеки в цепочке зависимостей. Менеджеры пакетов появились как раз для того, чтобы избавить разработчика от этой рутины. Они используются для автоматизации процесса установки пакетов, гарантируют, что все пакеты с зависимостями будут установлены, и предоставляют разумный механизм обновления пакетов при появлении новых версий.

В типичном приложении ASP.NET Core используются как минимум два менеджера пакетов: NuGet для пакетов .NET и другой менеджер для пакетов на стороне клиента (таких, как JQuery и Bootstrap). Также существуют другие варианты клиентских менеджеров пакетов; они будут рассмотрены в этой главе.

NuGet

NuGet — менеджер пакетов для .NET с открытым кодом. Он распространяется с Visual Studio и глубоко интегрируется в интерфейс командной строки dotnet. На момент написания книги NuGet управляет более 60 000 пакетов, общее количество загрузок которых превысило 1,3 миллиарда. Эти показатели ежедневно растут. Даже сборки, образующие ASP.NET, распространяются через NuGet, так что вам ничего не придется делать, чтобы начать пользоваться NuGet, — программа уже встроена.

Установка пакетов в NuGet

Существует несколько разных способов установки пакетов NuGet в приложениях ASP.NET Core. Первый вариант — редактирование секции `dependencies` в файле `project.json` приложения. Например, в файле `project.json` приложения `AlpineSkiHouse.Web` указана зависимость от версии 2.1.0 пакета `MediatR`.

```
зависимости из файла project.json приложения AlpineSkiHouse.Web
"dependencies": {
    // Другие зависимости опущены для краткости
    "MediatR": "2.1.0",
    "MediatR.Extensions.Microsoft.DependencyInjection": "1.0.0"
}
```

Чтобы установить пакеты из командной строки, выполните команду `dotnet restore`; эта команда устанавливает все пакеты, перечисленные в файле `project`.

json. Visual Studio автоматически выполняет команду `dotnet restore` при сохранении `project.json` после внесения изменений.

Инструментарий Visual Studio

Список пакетов NuGet для проекта отображается в узле `Reference` окна `Solution Explorer` (рис. 16.1).

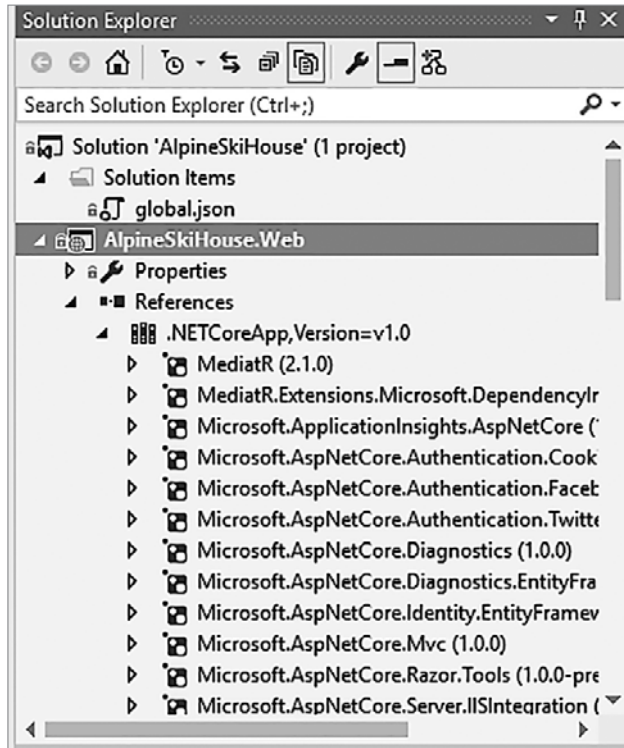


Рис. 16.1. Список пакетов NuGet в приложении `AlpineSkiHouse.Web`

Visual Studio также предоставляет полнофункциональный визуальный интерфейс для управления пакетами NuGet. Чтобы управлять пакетами NuGet для одного проекта, щелкните правой кнопкой мыши на проекте в `Solution Explorer` и выберите команду `Manage NuGet Packages`. Чтобы управлять пакетами NuGet для всего решения, щелкните правой кнопкой мыши на узле `Solution` в `Solution Explorer` и выберите команду `Manage NuGet Packages For Solution`.

Инструментарий `NuGet Package Manager` в Visual Studio (рис. 16.2) позволяет просматривать пакеты с `NuGet.org`, просматривать установленные пакеты и обновлять их.

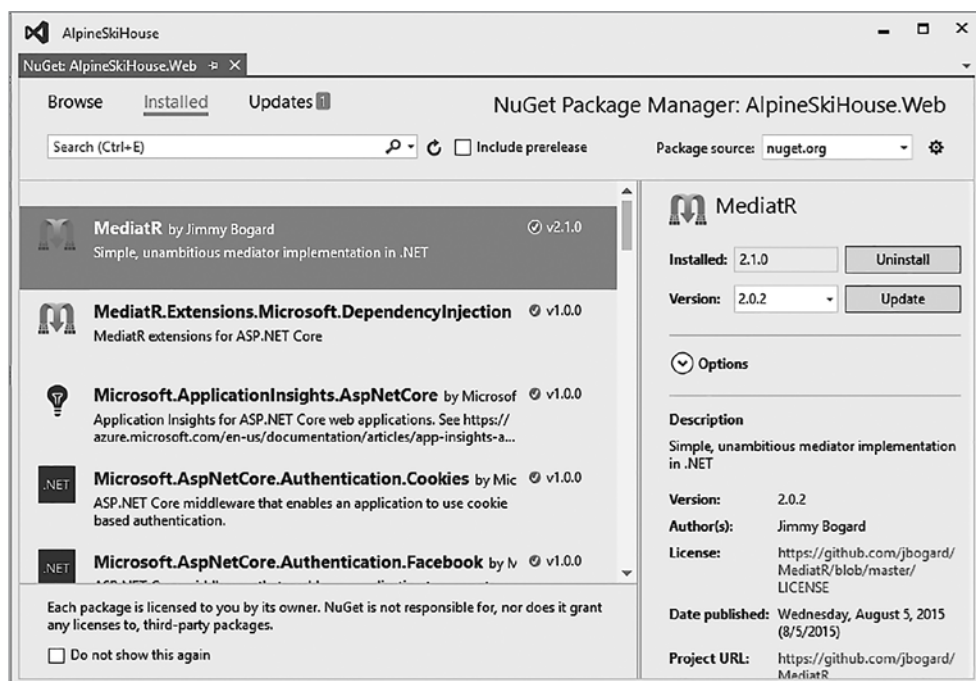


Рис. 16.2. NuGet Package Manager в Visual Studio

КАНАЛЫ NUGET

Канал NuGet представляет собой сетевой ресурс, в котором размещаются пакеты NuGet. Пакеты NuGet чаще всего берутся из канала nuget.org, но также могут загружаться из других источников. Некоторые крупные организации предпочитают ограничиться заранее утвержденным набором пакетов NuGet; в этом случае возможно создать внутренний канал NuGet, который содержит только утвержденные пакеты. Другой стандартный сценарий использования в компаниях — создание приватного канала NuGet для распространения внутренних программных пакетов организации.

Может показаться, что создание собственного канала NuGet — перебор, но делать это совсем несложно, и затраченные усилия часто окупаются. Ограничение списка разрешенных пакетов в организации может предотвратить загрузку разработчиком пакета, который заведомо создает проблемы или не обеспечивает должной поддержки или обновлений. Существует огромное количество решений, поддерживающих эти типы сценариев. MyGet (<http://myget.org/>) — популярный сервис с размещением пакетов, используемый командой ASP.NET для хоста ночных сборок. По адресу <https://docs.nuget.org/Contribute/Ecosystem> можно найти хороший обзор некоторых из наиболее популярных продуктов.

Многие варианты, перечисленные в этой статье, также работают с другими менеджерами пакетов — такими, как `npm` и `bower`.

npm

`npm` — менеджер пакетов для Node.js. Возможно, вы спрашиваете себя: «Почему мы говорим о Node.js в книге, посвященной написанию приложений ASP.NET Core?» Если взглянуть на веб-разработку в более широком контексте, многие инструменты для веб-разработчиков пишутся в виде пакетов Node.js и распространяются с использованием `npm`. Примеры таких инструментов — компилятор TypeScript, системы автоматизации выполнения задач `gulp` и `Grunt`, модульная инфраструктура `Jasmine`. Как было показано в главе 15 «Роль JavaScript», язык JavaScript играет важную роль во всех современных веб-приложениях. Доступность инструментов, используемых разработчиками за пределами мира ASP.NET, важна хотя бы для того, чтобы оставаться в контакте с большим сообществом веб-разработки.

`npm` — инструмент командной строки. Чтобы использовать `npm` в своем приложении, сначала создайте файл `package.json`. Это можно сделать вручную или же воспользоваться командой `npm init`, которая задаст несколько вопросов и создаст файл `package.json` автоматически. Ниже приведен базовый файл `package.json` со ссылками на фреймворк модульного тестирования `Jasmine` и систему выполнения модульных тестов `Karma`. Настройка `Jasmine` и `Karma` более подробно рассматривается в главе 20 «Тестирование».

```
{
  "name": "alpineskihouse",
  "version": "1.0.0",
  "description": "",
  "private": true,
  "main": "gulpfile.js",
  "dependencies": {},
  "devDependencies": {
    "jasmine": "2.5.0",
    "karma": "1.2.0"
  },
  "author": "Alpine Ski House"
}
```

Используемые пакеты перечисляются в секции `devDependencies`, потому что эти пакеты не должны распространяться как часть нашего приложения, а лишь используются разработчиками в ходе работы над приложением. Также свойству `private` присваивается значение `true`, чтобы приложение не было случайно опубликовано в общедоступном реестре пакетов `npm`.

ПРИМЕЧАНИЕ

Команда `npm install` восстанавливает все зависимости, перечисленные в файле `package.json`. По умолчанию все перечисленные зависимости в секциях `dependencies` и `devDependencies` загружаются в текущий проект в папку с именем `node_modules`.

Добавление зависимостей

Новые зависимости могут добавляться прямым редактированием файла `package.json` или выполнением команды `npm install` с указанием имени пакета и флага `--save-dev`. Также обратите внимание на флаг `--save`, который добавляет пакеты в секцию `dependencies` файла `package.json`. Этот флаг используется для добавления зависимостей, необходимых во время выполнения.

```
npm install karma --save-dev
```

По умолчанию `npm` добавляет зависимость для последней стабильной версии заданного пакета. Если приложение требует конкретной версии пакета, либо отредактируйте файл `package.json` вручную, либо укажите версию, добавив конструкцию `@x.x.x` в имя пакета при выполнении команды `npm install`.

```
npm install karma@1.2.0 --save-dev
```

Использование модулей npm

Многие модули `npm` включают интерфейсы командной строки, чтобы их можно было использовать прямо из командной строки без входа в среду `Node`. Хороший пример — система запуска тестов `Karma` с интерфейсом командной строки `karma`, который может использоваться для запуска модульных тестов. Чтобы предоставить глобальный доступ к таким инструментам, как `karma`, следует установить их глобально на машине разработчика или билд-сервере (глобальная установка осуществляется с флагом `-g`).

```
npm install karma -g
```

Глобальная установка пакетов требует навыков более основательного конфигурирования, чем хотелось бы, особенно у начинающих разработчиков. В идеале начинающий разработчик должен просто выполнить команду `npm install` и немедленно получить доступ к необходимым командам. Чтобы это стало возможно, `npm` устанавливает инструменты командной строки локально в папке `/node_modules/.bin/`. Таким образом, даже если инструмент командной строки `Karma` недоступен в глобальном пути разработчика, он сможет обратиться к нему:

```
.\ node_modules\.bin\karma start
```

Чтобы задача стала еще проще, определите сценарий `npm` в файле `package.json`:

```
{
  "name": "alpineskihouse",
  "version": "1.0.0",
  "description": "",
  "private": true,
  "main": "gulpfile.js",
  "dependencies": {},
  "devDependencies": {
    "jasmine": "2.5.0",
    "karma": "1.2.0"
  },
  "author": "Alpine Ski House",
  "scripts": {
    "test-js": ".\\node_modules\\.bin\\karma start --single-run"
  }
}
```

После того как сценарий будет определен, его можно выполнить командой `npm run-script`:

```
npm run-script test-js
```

Интеграция с Visual Studio

Инструментарий командной строки `npm` выглядит знакомо, но пользователь интегрированной среды Visual Studio (IDE), вероятно, предпочтет более высокую степень интеграции взаимодействий. К счастью, последние версии Visual Studio отлично интегрированы с `npm`. Visual Studio автоматически обнаруживает, когда проект содержит файл `package.json`, и включает `npm` в узел **Dependencies** в окне Solution Explorer (рис. 16.3).

Visual Studio автоматически обнаруживает изменения в файле `package.json` и выполняет команду `npm install` при необходимости. Вы также можете вручную инициировать выполнение `npm install` — щелкните правой кнопкой мыши на узле `npm` в Solution Explorer и выберите в меню команды **Restore Packages**. Чтобы просмотреть вывод команды `npm install` в окне вывода Visual Studio, выберите режим вывода `bower/npm`.

ПРИМЕЧАНИЕ

При использовании расширения Visual Studio можно выполнять сценарии `npm` прямо из Visual Studio при помощи окна Task Runner Explorer. Сначала выполните указания на странице <https://github.com/madskristensen/NpmTaskRunner>, чтобы установить расширение NPM Task Runner. Затем откройте окно Task Runner Explorer в Visual Studio командой **View ▸ Other Windows ▸ Task Runner Explorer**. В окне Task Runner Explorer перечислены все сценарии `npm` текущего проекта. Чтобы выполнить отдельную задачу, щелкните на ней правой кнопкой мыши и выберите команду **Run** (рис. 16.4).

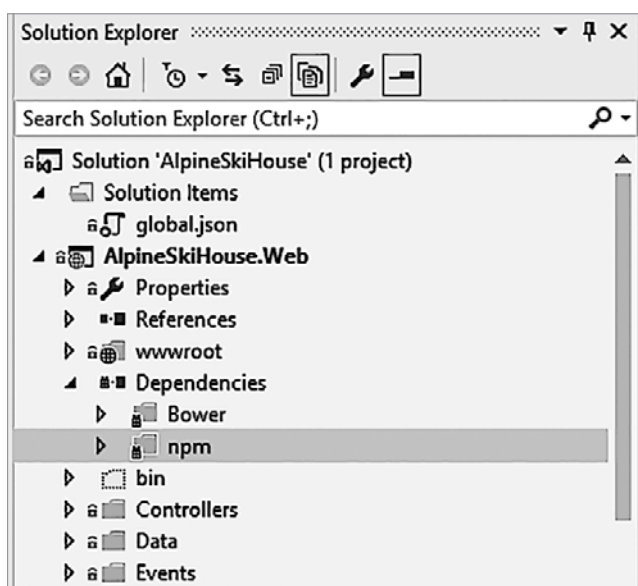


Рис. 16.3. Узел зависимости npm в Solution Explorer

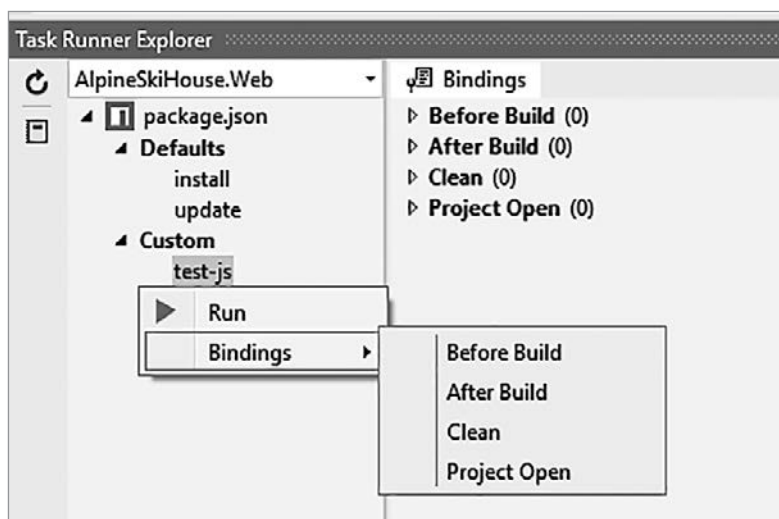


Рис. 16.4. Добавление привязок к сценариям npm в окне Task Runner Explorer

Task Runner Explorer также предоставляет возможность привязки сценариев к определенным событиям Visual Studio. Например, сценарий test-js можно настроить на выполнение перед событием Build.

Yarn

Основным менеджером пакетов для средств JavaScript на стороне сервера является npm. Некоторые разработчики даже используют его для установки клиентских библиотек (ниже будет рассказано о bower, альтернативном менеджере библиотек на стороне клиента). Тем не менее у npm есть свои недостатки. В этом разделе мы рассмотрим npm и Yarn — новый менеджер пакетов JavaScript, который разрабатывался для устранения всех недостатков npm. Возьмем файл `package.json`, в котором указаны версии всех модулей, включенных в npm. Если вас беспокоит, что у участников вашего проекта окажутся неправильные версии библиотек, номер версии пакета можно ограничить:

```
"devDependencies": {
  "@types/chartjs": "0.0.28",
  "@types/lodash": "^4.14.34",
}
```

Пакету `@type/chartjs` назначается конкретный номер версии; это означает, что устанавливается всегда именно эта версия пакета. С другой стороны, для пакета `@types/lodash` версия начинается с символа `^`. Этот символ означает, что npm должен установить версию `@types/lodash`, совместимую с указанной, согласно семантическому управлению версиями (semver). При восстановлении пакета в момент публикации вы получите версию `@types/lodash@4.14.37`. Даже при указании точного номера версии в файле `packages.json` абсолютное совпадение установок не гарантировано. Пакеты могут иметь свои зависимости, а их авторы могли использовать неконкретные версии. Например, случайный файл `package.json` из каталога `node_modules` содержит следующую информацию:

```
"devDependencies": {
  "chai": "^3.2.0",
  "coveralls": "^2.11.2",
  "graceful-fs": "4.1.4",
  "istanbul": "^0.3.20",
  "mocha": "^2.0.0",
  "rimraf": "^2.4.3",
  "sinon": "^1.10.3",
  "sinon-chai": "^2.6.0"
},
```

Почти все пакеты содержат ограничения по семантическому управлению версиями. Кроме того, работа npm не детерминирована; это означает, что разный порядок установки пакетов может привести к установке разных версий модулей у пользователей, выполняющих операцию установки. И это довольно серьезная проблема, потому что фрагмент кода, работающий на рабочей станции А, может не работать на рабочей станции Б из-за различий в версиях пакетов.

Полное восстановление каталога `node_modules` из интернета может быть ужасно медленным. Оно потребует загрузки и распаковки буквально тысяч пакетов. На билд-сервере это может привести к кардинальному замедлению процесса сборки. Что еще хуже, случайные сбои в ходе установки (например, перебои в работе сети) способны полностью нарушить процесс сборки.

Также каталог `node_modules` можно сохранить в репозитории, но он содержит тысячи файлов, обновление пакета обычно создает колоссальные изменения в истории `git`, а это нежелательно. Проблема в том, что `npm` не создает локальный кэш пакетов, поэтому каждая сборка требует обращения к реестру пакетов.

В прошлом эти дефекты `npm` приходилось попросту терпеть, но с появлением менеджера пакетов `Yarn` это время прошло. Новый менеджер пакетов можно подставить на место `npm` с минимальными изменениями. `Yarn` — проект Facebook, Exponent, Google и Tilde, стремящийся преодолеть основные недостатки `npm`. Прежде всего `Yarn` обеспечивает детерминированную установку пакетов в проектах. В каком бы порядке ни устанавливались пакеты, дерево пакетов в `node_modules` останется неизменным. `Yarn` также выполняет параллельные загрузки и установки с повторными попытками в случае неудачи. Таким образом ускоряется процесс установки, а его надежность повышается.

Чтобы установить `Yarn`, достаточно выполнить следующую команду:

```
npm install -g yarn
```

Теперь `Yarn` может использоваться с существующим файлом `package.json`. Удалите каталог `node_modules` и введите команду:

```
yarn
```

Для добавления пакетов уже не нужно указывать флаг `--save` или `--save-dev`, потому что они будут сохраняться автоматически. При добавлении флага `--dev` пакет будет сохранен в секции `devDependencies`:

```
yarn add jquery  
yarn add gulp --dev
```

Секрет возможностей `Yarn` кроется в новом файле с именем `yarn.lock`. Этот файл должен быть включен в систему управления версиями, чтобы все пользователи получили одинаковые версии. Фрагмент файла выглядит так:

```
mkdirp@^0.5.0, mkdirp@^0.5.1, "mkdirp@>=0.5.0", mkdirp@~0.5.0, mkdirp@~0.5.1,  
mkdirp@0.5:  
  version "0.5.1"  
  resolved "https://registry.yarnpkg.com/mkdirp/-/mkdirp-0.5.1.tgz#30057438eac6cf7f  
    8c4767f38648 d6697d75c903"  
  dependencies:  
    minimist "0.0.8"
```

В этом фрагменте перечислены версии, которые должны использоваться для пакета с именем `mkdirp`. В первой строке перечислены версии `mkdirp`, каждая из этих версий упоминается где-то в дереве пакетов. Yarn обнаруживает все эти версии и преобразует их в одну версию, которая может удовлетворить все требуемые версии.

Yarn справляется со многими недостатками `npm`. Тем не менее еще рано судить о том, получит ли Yarn более широкое распространение. Простота перехода с `npm` на Yarn, возрастание скорости и детерминированная установка пакетов выглядят довольно привлекательно, поэтому программа скорее будет принята, чем нет.

bower

ASP.NET Core использует `bower` как менеджер пакетов по умолчанию для библиотек на стороне клиента. Такие библиотеки состоят из ресурсов, которые в конечном итоге передаются браузеру клиента. Обычно такими ресурсами являются JavaScript и CSS, как и в случае с jQuery и Bootstrap, но также возможны и другие виды ресурсов: изображения, шрифты и значки. В крупном веб-приложении вполне могут использоваться десятки клиентских библиотек, поэтому важно иметь в распоряжении хороший менеджер пакетов.

ПОЧЕМУ ИМЕННО BOWER?

В предыдущих версиях ASP.NET для распространения клиентских библиотек использовался NuGet, но, к сожалению, для управления пакетами на стороне клиента NuGet подходит плохо. Серьезная проблема заключается в том, что библиотеки создаются и используются более широким сообществом веб-разработки. Так как сообщество разработки .NET составляет лишь небольшую часть сообщества в целом, авторы клиентской библиотеки могут не иметь опыта работы с .NET, и от них нельзя ожидать знания всех тонкостей NuGet или сопровождения пакетов NuGet именно для разработчиков .NET. Во многих случаях разработчик-энтузиаст из сообщества Microsoft пытается заниматься сопровождением пакета NuGet от имени авторов библиотеки, но такая практика оказалась ненадежной и тягостной. В результате пакеты NuGet для многих популярных библиотек на стороне клиента оказывались либо недоступными, либо безнадежно устаревшими.

Вместо того чтобы пытаться сопровождать клиентские пакеты в NuGet, в ASP.NET Core и Visual Studio была реализована превосходная поддержка менеджеров пакетов, часто используемых в сообществе веб-разработчиков. `bower` — один из таких менеджеров пакетов — используется в шаблонах приложений ASP.NET Core по умолчанию.

В Visual Studio имеются превосходные инструменты, которые скрывают от разработчика часть сложностей `bower`, но для начала посмотрим, как `bower` используется из командной строки. `bower` — программа командной строки на базе Node, устанавливаемая через `npm`. Для глобальной установки используется следующая команда:

```
npm install bower -g
```

При работе с bower пакеты определяются в файле `bower.json`. Этот файл создается вручную или командой `bower init`.

```
{
  "name": "alpineskihouse",
  "private": true,
  "dependencies": {
    "bootstrap": "3.3.6",
    "jquery": "2.2.0",
    "jquery-validation": "1.14.0",
    "jquery-validation-unobtrusive": "3.2.6"
  }
}
```

Имя "alpineskihouse" используется для идентификации текущего пакета, а установка флага `private` гарантирует, что приложение не будет случайно опубликовано в реестре пакетов bower. Авторы библиотек должны присвоить флагу `private` значение `false`. Библиотеки, используемые приложением, перечисляются в секции `dependencies`. С каждой библиотекой также связывается версия. Чтобы установить все пакеты, перечисленные в файле `bower.json`, выполните команду `bower install`:

```
bower install
```

По умолчанию все перечисленные зависимости загружаются в папку с именем `bower_components` текущего проекта. Чтобы изменить место хранения, укажите другой каталог в файле конфигурации `.bowerrc`. В веб-проекте Alpine Ski House каталог по умолчанию был заменен каталогом `wwwroot/lib`:

Содержимое файла `.bowerrc`

```
{
  "directory": "wwwroot/lib"
}
```

Добавление зависимостей

Чтобы добавить зависимость для нового пакета, либо отредактируйте файл `bower.json` вручную, либо выполните команду `bower install` с указанием имени пакета и флага `--save`.

```
bower install jquery --save
```

Как и `npm`, bower добавляет зависимость для последней стабильной версии заданного пакета. Если приложение требует конкретной версии пакета, укажите флаг `--save-exact`.

```
bower install jquery --save --save-exact 2.4.1
```


bower извлекает пакеты из общедоступного репозитория git, обычно GitHub. Время от времени требуется использовать предварительную версию библиотеки, которая еще не стала итоговой; благодаря возможности загрузки пакетов с GitHub это выполняется без малейших проблем. Вы можете приказать bower обратиться к конкретному репозиторию git и даже к конкретному сохранению в репозитории. Чтобы эти переопределения стали возможными, укажите для зависимости URL-адрес git вместо номера версии:

```
"canvg": "https://github.com/gabelerner/canvg.git#f5519bc910ecefe083f636707b27e436386fdeed"
```

Обращение к ресурсам из пакетов bower

После выполнения команды `bower install` необходимые вам ресурсы находятся в каталоге компонентов bower (в нашем случае это каталог `wwwroot/lib`). К этим сценариям можно обращаться в приложениях точно так же, как это делается с любыми другими сценариями или таблицами стилей. Местонахождение файлов зависит от конкретного пакета bower, но многие пакеты соблюдают правило, согласно которому важные файлы размещаются в папке с именем `dist`.

```
<html>
  <head>
    ...
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    ...
  </head>
  <body>
    ...
    <script src="~/lib/jquery/dist/jquery.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
    ...
  </body>
</html>
```

В рабочей среде стоит брать некоторые распространенные клиентские библиотеки из CDN (Content Delivery Network), а не из версий, хранящихся на вашем веб-сервере. ASP.NET предоставляет тег-хелперы, упрощающие этот процесс. Тема тег-хелперов будет рассматриваться в главе 19 «Многоразовый код».

Итоги

Ситуация в области веб-разработки непрерывно изменяется. Хотя bower в наши дни пользуется популярностью как менеджер пакетов ASP.NET по умолчанию для библиотек на стороне клиента, существуют и другие варианты, которые со

временем могут выйти на первый план. Некоторые разработчики используют прт как для инструментов разработчика, так и для клиентских библиотек; JSPM — еще одна альтернатива bower, набирающая популярность. Даже в мире .NET, в котором NuGet занимает ведущие позиции, появляются альтернативные менеджеры пакетов, такие как Paket.

Выбор менеджера пакетов для приложения Alpine Ski House определялся как шаблонами проектов ASP.NET Core по умолчанию, так и опытом участников группы. В конечном итоге большинство менеджеров пакетов предоставляет сходную функциональность. Выберите то, что подходит для вашей команды, и не бойтесь рассматривать альтернативные решения, если вы считаете, что они могут упростить процесс разработки для вашей команды и вашего приложения. В следующей главе речь пойдет о том, как использовать таблицы стилей, для того чтобы интерфейсная часть приложения выглядела эффектно.

17

Стильный интерфейс

Марк-2 сообщил Даниэль о регрессии, которая успела встретиться ему пару раз. Он исправил ее на своей машине, но после того, как последняя сборка отправилась на промежуточный сервер, изменения куда-то пропали.

«Видишь? — начал Марк-2. — Вот здесь я сохранил свое изменение!»

«Да, что-то странное, — ответила Даниэль. — Ты уверен, что твои изменения добрались до билд-сервера?»

«Да. Уверен на сто процентов. Я проверил контрольную сумму SHA, и она в точности соответствует изменениям перед сборкой. И все равно — посмотри на разметку CSS, которая была развернута...»

«Одну минуту! — перебил Адриан. — Ты говоришь о сохранении изменений в таблицах стилей?»

«Да, — ответил Марк-2. — Клянусь, этот запрос возвращается ко мне уже в третий раз. Я вношу все изменения, локально все работает, и после сохранения в репозитории я убеждаюсь в том, что оно на месте».

«Он прав, — добавила Даниэль. — Я проверяла его код при последнем запросе на включение изменений. Все изменения были внесены. Но все выглядит так, словно после слияния происходит перезапись или что-то в этом роде. Мне даже не удалось понять, где это происходит».

Адриан широко улыбнулся: «А я знаю». Он снова плюхнулся в кресло и подкатился к Марку-2 и Даниэль. Вид у него был возбужденный, и он явно ждал, кто из них спросит, что же происходит.

«Ладно, Адриан, — рассмеялась Даниэль. — Хватит ухмыляться, помоги нам. Из-за чего происходит перезапись?»

«Я так рад, что вы спросили, — он начал довольно потирать ладони. — Хорошо, вы помните, что я на этой неделе работал в паре с другим Марком? Мы как раз этим занимались».

«Устраивали заговор по отмене моих изменений в CSS?» — предположил Марк-2.

«В общем-то да... в своем роде, — ответил Адриан. — Наверное, это наша вина, потому что мы не обновили `git.ignore`, но CSS в репозитории теперь быть не должно».

«Постой, но если все хранится в системе управления исходным кодом, то почему там не должно быть CSS? — спросила Даниэль. — Я пропустила какое-то собрание?»

«Потому что теперь CSS у нас становится артефактом сборки. — На физиономию Адриана вернулась прежняя широкая ухмылка. — Вы не представляете, как меня это радует. Вы меня здесь многому научили, но вот прошел месяц — и мне тоже удалось вам что-то объяснить».

«Да, да, Адриан, — сказал Марк-2. — Я тебе сделаю медаль. — Все рассмеялись. — А теперь признавайся, что происходит?»

«Леди и джентльмены, позвольте обратить ваше внимание на файл SCSS, который теперь хранится в папке Style вашего любимого проекта...»

Создание сайтов с таблицами стилей

Когда мы впервые решили написать главу о стилях, предполагалось, что писать мы будем о CSS-фреймворке Bootstrap. В конце концов, Bootstrap доминирует в мире .NET и включается в шаблоны по умолчанию с 2013 года. Однако это достаточно зрелый фреймворк, и существует много ресурсов, посвященных тому, как интегрировать его в приложения, поэтому в этой главе мы решили сосредоточиться на процессе. Речь пойдет о том, как команда Alpine использовала разметку CSS и встроила ее в свой проект. Мы поделимся альтернативными стратегиями организации рабочего процесса и даже рассмотрим некоторые альтернативы для Bootstrap. Тем не менее из-за своей зрелости Bootstrap может использоваться как ориентир для моделирования некоторых целей, которых нам хотелось бы достичь при описании стилей для нашего сайта.

Концепция каскадных таблиц стилей, или CSS (Cascading Style Sheet), существует почти столько же времени, сколько существуют браузеры. В этой главе мы не будем уговаривать вас использовать CSS, потому что каждому, кто занимался сколько-нибудь серьезной веб-разработкой за последние 20 лет, известно о таблицах стилей и их влиянии на разработку. С другой стороны, если вы принадлежите к числу новичков, вы с большой вероятностью были вовлечены в использование CSS без особых размышлений. А если вы не используете таблицы стилей... что ж, вы либо идете по пути, который был коллективно признан ошибочным, либо просто не занимаетесь созданием сайтов.

Как минимум нам хотелось бы подтвердить то, что вам может быть уже известно; пролить свет на то, почему что-то делается именно так, а не иначе; и представить некоторые возможности использования средств поддержки CSS в мире ASP.NET Core MVC.

Немного истории

Корни CSS уходят в область маркетинга — в те времена, когда специалисты, ответственные за успех рекламных кампаний, пришли к довольно важным выводам. Если вы руководствуетесь стилевыми рекомендациями, аудитория лучше воспринимает информацию, лучше запоминает ваш фирменный стиль и сразу определит, что это ваша кампания, по самым неприметным элементам. За примерами далеко ходить не надо — достаточно вспомнить крупнейших производителей колы или сети фастфуда; простой логотип и даже буква фирменного шрифта вызывают ассоциации с брендом в целом.

Напротив, визуально непривлекательную информацию труднее связать с определенным брендом. Даже реклама на целый разворот с бедным оформлением будет восприниматься плохо. Задача дизайнера — позаботиться о том, чтобы способ передачи информации не отвлекал от самого информационного наполнения.

В основе стилевых руководств, стилевых рекомендаций или таблиц стилей на веб-языке лежит одна идея: построение целостного набора правил, влияющих на форматирование контента. Речь идет об отделении контента от представления так, чтобы визуальные цели дизайна были видны и последовательно применялись для передачи информации в любом виде, будь то печатные издания, телевидение или веб-сайт.

Спецификация CSS была предложена Хоконом Виумом Ли (Håkon Wium Lie) в 1996 году, но до появления ее полноценной поддержки в браузерах прошло почти четыре года. И хотя спецификация избавляла разработчика от необходимости явно назначать стили каждому элементу для соблюдения правил по шрифтам, цветам, размерам и обрамлению, ранние реализации были непоследовательны, а основная идея с отделением представления от контента с трудом воплощалась на практике.

CSS состоит из нескольких простых концепций. Имеется синтаксис описания правил, который должен быть понятен клиенту, выполняющему отображение документа. Селекторы определяют целевые элементы вашего документа и назначают правила отображения этих элементов. Наконец, существует концепция присваивания: элементам, соответствующим определяемым правилам, присваиваются значения, выраженные в единицах различных типов.

Трудно представить, чтобы хоть кто-то из обитателей пространства веб-разработки не работал с CSS в том или ином качестве. Впрочем, хотя CSS существует почти два десятилетия, нельзя отрицать, что работать с CSS не всегда было легко. Стандарт продолжал развиваться, пока коммерческие организации пытались использовать его в своих веб-приложениях. Разные фирмы-разработчики по-разному интерпретировали спецификации CSS, в результате их реализации создавали разные макеты. Это приводило к пересмотрам спецификации, которые временно нарушали оформление существующих сайтов. С годами ситуация радикально

улучшилась, но даже когда два браузера заявляют о том, что они реализуют некоторую функцию, в окончательном продукте вы можете увидеть разные результаты. Этот факт преследовал разработчиков с того момента, когда был выпущен второй браузер с поддержкой CSS.

Причины, по которым мы используем CSS, сформулировать несложно. Если бы вы решили объяснить это человеку, не имеющему отношения к миру веб-разработки, объяснение выглядело бы примерно так:

«Представьте, что я создаю для вас веб-сайт — это необходимо, чтобы ваш бизнес не отставал от конкурентов. У вас есть некие брендовые элементы, которые вы хотите видеть на своем сайте; вы выбираете цвета, шрифты и сообщаете другую нужную информацию. Потом вы пишете весь контент для сайта и передаете мне сорок страниц текста, которые превращаются в десятки веб-страниц. На каждой странице используются одни и те же шрифты, одни и те же цвета и интервалы, имитирующие печатную документацию, которую вы раздаете своим клиентам. Каждая из десятков создаваемых страниц содержит сотни элементов: абзацы, столбцы, кнопки, ссылки на другие страницы меню и т. д., и все они должны быть оформлены в едином стиле. Чтобы все эти тысячи элементов соответствовали вашим рекомендациям, мне нужен инструмент для описания правил, которым должны соответствовать элементы. Этот инструмент называется каскадными таблицами стилей, или CSS (Cascading Style Sheet). Если через месяц вы решите, что вместо светло-синих рамок должны использоваться темно-синие, я могу либо внести одно изменение в CSS, либо изменять все элементы с рамками на всех страницах. Кстати, оплата у меня почасовая. Решайте сами».

Старайтесь не обращать особого внимания на номера версий, особенно когда вам попадают обозначения вида CSS3 или CSS4 — с технической точки зрения они вообще не существуют¹. Когда-то рабочая группа CSS из комитета WC3 осознала, что вся спецификация CSS не может двигаться вперед как единое целое, и поэтому переименовала все, что появилось после CSS 2.1, в CSS3. Все компоненты CSS теперь представляются в виде модулей; сейчас их 35, и их количество продолжает расти. Такой подход оказался в высшей степени плодотворным для сообщества разработчиков по двум причинам. Во-первых, проблемы с прояснением отдельных модулей (например, фонами или рамками) решались намного быстрее, неоднозначности в правилах устранялись, а визуализация в разных браузерах становилась более целостной. Во-вторых, появлялась возможность введения новых модулей еще до того, как рабочая группа согласует все аспекты CSS в целом.

ПРИМЕЧАНИЕ

Не пытайтесь запоминать, какие модули CSS поддерживаются теми или иными браузерами и какие именно функции модулей поддерживаются. Лучше загляните на сайт <http://caniuse.com>; здесь вы сможете выполнять поиск по браузеру, версии, странам и многим другим характеристикам, чтобы определить, какие функции подойдут для вашего проекта.

¹ <http://www.xanthir.com/b4Ko0>.

В листинге 17.1 приведен пример упрощения одного элемента за счет использования таблицы стилей. В первой строке условного документа HTML применяется встроенное стилевое оформление, а во второй для назначения атрибутов стиля включено имя класса.

Листинг 17.1. Применение CSS для упрощения тега HTML

```
<div style="border: 1px solid blue; padding: 10px; color: navy;">hello!</div>
<div class="simple-label victory">hello!</div>
```

На первый взгляд различия незначительны, но даже на этом примере с одним элементом можно представить изменения, которые потребуются в описанной выше ситуации с заказчиком. Изменения в `simple-label` или `victory` могут быть централизованы, и с изменением в одном месте вы сможете обновить каждый элемент на всех страницах, использующих эти классы.

Создание собственной таблицы стилей

Таблицы стилей представляют собой обычные текстовые файлы — почти такие же, как все остальные файлы с исходным кодом в вашем проекте.

Чтобы создать файл CSS, создайте текстовый документ с расширением `.css`. Visual Studio также поддерживает возможность добавления документов CSS из диалогового окна `Add New Item` (рис. 17.1).

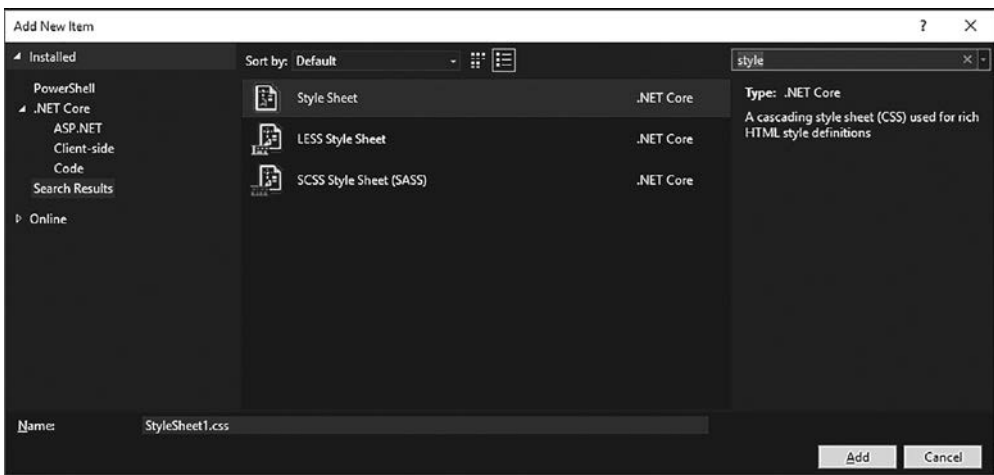


Рис. 17.1. Диалоговое окно `Add New Item` при добавлении таблицы стилей

Чтобы включить таблицу стилей в страницу, добавьте строку разметки следующего вида:

```
<link rel="stylesheet" href="~/css/site.css" />
```

После того как документ CSS будет создан и включен в страницу, определите правила оформления, которым должна соответствовать страница. Правила состоят из двух частей (рис. 17.2): в большом прямоугольнике находится блок селекторов, а в маленьком — блок объявления.

```
input,  
.sidebar-column,  
#news-letter {  
    max-width: 280px;  
}
```

Рис. 17.2. Блоки селектора и объявления

Блок селекторов может содержать любое количество селекторов. Селектором может быть элемент HTML (например, `input`, как в приведенном примере), имя класса (`.sidebar-column` в примере) или уникальный идентификатор элемента на странице (`#newsletter`). Селекторы также могут включать условия присутствия или отсутствия, совпадений подстрок, иерархии и наследования, а в особом случае так называемые псевдоселекторы описывают состояние элемента или его отношение с другим элементом (например, посещенную ссылку или то, предшествует ли элемент другому элементу). Включите столько селекторов, сколько нужно. В объявлении правила задаются значения свойств; присвоенные значения будут применены ко всем элементам, совпадающим с селекторами из блока селекторов.

Все правила применяются к элементам документа определенным образом. Выбор стиля для конкретного элемента зависит от порядка определения правил, уровня вложенности элемента в документе и порядка применения стилей классов к элементам. При этом используется концепция специфичности (*specificity*); впрочем, ее рассмотрение выходит за рамки этого раздела.

За информацией об определении селекторов и специфичности обращайтесь на сайт W3C по адресу <https://www.w3.org/TR/css3-selectors/>.

В ASP.NET Core MVC вы, скорее всего, предпочтете поместить таблицы стилей в макет, чтобы стиль мог совместно использоваться несколькими представлениями. Как упоминалось в главе 11 «Представления Razor», вы должны включить разные версии своей таблицы стилей для разных сред. Тег-хелпер `environment` позволяет включить сокращенную версию CSS в рабочую версию, при этом сохранив удобочитаемую версию в локальной среде разработки.

```
<environment names="Development">  
    <link rel="stylesheet" href="~/css/site.css" />  
</environment>  
<environment names="Staging,Production">  
    <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />  
</environment>
```


Здесь снова проявляется изящество Razor: для определения того, какая таблица стилей должна использоваться в конкретном случае, можно подключать условные конструкции и условные команды. Если у вас имеется приложение с возможностью стиливого оформления на базе источников CSS, задаваемых пользователем, перед вами открывается замечательная возможность динамического изменения внешнего вида всего сайта всего в нескольких строках кода.

Создание разметки CSS может быть достаточно неудобным, особенно если вы пытаетесь предусмотреть разные варианты реализации отдельных аспектов спецификации в разных браузерах. Создание набора правил, распространяющихся на несколько элементов или включающих префиксы браузеров — монотонное и ненадежное дело. Это особенно справедливо, потому что общий дизайн, который вы создаете, часто содержит повторяющиеся элементы, блоки с похожими объявлениями правил и зависит от иерархий документа, формирующих опыт пользовательского взаимодействия.

Как всегда бывает в области разработки приложений, при возникновении проблем кто-то строит инструментарий для их решения.

SASS в работе со стилями

За прошедшие годы стало ясно, что написание всей разметки CSS вручную (а это дело крайне монотонное) превращается в рутину, которую можно было бы упростить. Что еще важнее, документы CSS разбивались на меньшие, более удобные фрагменты, и разработчики хотели исключить вероятность появления ошибок из-за несовпадения цветов, шрифтов и размеров. В CSS отсутствует возможность составления документов, определения и совместного использования переменных, применения логики и импортирования других правил. Если мы стремимся к исключению монотонной работы с высоким риском получения ошибок из документов HTML, зачем останавливаться на уровне CSS и подвергать себя всем этим рискам?

Также вполне очевидно, что для упрощения процесса разработки изменять спецификацию синтаксиса не нужно. CSS может предоставлять метаданные для элементов HTML, но CSS не является языком программирования и не должен эволюционировать по правилам языков программирования. В то же время CSS не предоставляет достаточных средств для преодоления некоторых недостатков, и не каждому захочется писать 500 строк кода для того, чтобы определить стили кнопок вручную.

С учетом всего сказанного появился специальный класс инструментов, называемых «препроцессорами CSS». Их целью является расширение CSS с возможностью применения синтаксиса программирования для построения стиливых документов с сохранением стопроцентной совместимости с CSS. В препроцессорах поддерживаются переменные, операторы, наследование и функции, чтобы дизай-

неры и проектировщики могли работать с CSS так же, как мы работаем с HTML. Даже можно сказать, что писать CSS вообще больше не придется — мы пишем «программы», которые генерируют CSS за нас. В табл. 17.1 перечислены некоторые популярные препроцессоры, представленные на рынке.

Таблица 17.1. Популярные препроцессоры CSS

Препроцессор	Преимущества	Место в разработке .NET
SASS, SCSS	Пожалуй, самый зрелый из всех препроцессоров; отличная документация, популярность, использование в ведущих фреймворках CSS	Возможно создание как из режима командной строки, так и из встроенного инструментария Visual Studio; хороший набор функциональных возможностей для большинства потребностей разработки CSS
LESS	Возможность использования исходного кода LESS с препроцессором JS в браузере	Возможно создание как из режима командной строки, так и из встроенного инструментария Visual Studio; препроцессор достаточно зрел, чтобы пользоваться доверием, но некоторые фреймворки CSS от него отходят
Stylus	Автоматическое использование префиксов фирм-разработчиков, свыше 60 встроенных функций, обширная функциональность	Требует node.js и некоторых настроек для компиляции в среде Visual Studio. Потенциально требует более высоких усилий при изучении, но более эффективен при самостоятельном построении сложных фреймворков CSS

Кроме этих препроцессоров, существует много других; одни требуют PHP, другие обеспечивают более качественную поддержку разработки Ruby и Rails, третьи находятся на стадии становления, но стремятся заполнить пробелы, остающиеся у рыночных лидеров. Все они используют похожие конструкции, предоставляющие разработчикам большую гибкость в создании таблиц стилей. Проблемы в проекте Alpine возникли именно из-за компиляции файла SCSS, потому что билд-сервер перезаписывал файл CSS в ходе работы.

SASS (Syntactically Awesome Style Sheets) — предшественник SCSS. Если SASS ориентируется на более лаконичный синтаксис с отступами и игнорированием фигурных скобок и символов «;», SCSS является надмножеством CSS — это означает, что каждый действительный документ CSS также является действительным документом SCSS. Благодаря этому факту группы разработчиков могут переключаться на SCSS с минимальными затратами усилий и немедленно начинать пользоваться усовершенствованиями SCSS.

Основы SCSS

Добавить в проект файл SCSS ничуть не сложнее, чем файл любого другого типа. Взгляните на рис. 17.1: в диалоговом окне создания файла присутствует вариант создания файла SCSS вместо простого файла CSS. Выберите этот вариант или просто переименуйте файл CSS, над которым вы работаете.

Переменные

Начнем с одной из самых серьезных претензий, которые приходится слышать от разработчиков при работе со стилевыми таблицами: это необходимость их обновления. Представьте следующую ситуацию: вы работаете с палитрой и получаете список цветов от дизайнера. Вместо того чтобы использовать эти цвета непосредственно в правилах, вы создаете переменные так, как показано в листинге 17.2.

Листинг 17.2. Определение цветов как переменных в SCSS

```
$dark-border:      #7a7265;
$light-background: #cecac4;
$main-text-color:  #433e3f;
$primary-border-width: 2px;
```

С такими переменными к цветам можно обращаться по именам вместо шестнадцатеричных кодов, как в листинге 17.3. Преимущества такого решения очевидны: чтобы изменить все значения переменной, вам не придется просматривать весь код или пользоваться средствами поиска/замены в редакторе.

Листинг 17.3. Использование переменной в правилах CSS

```
.banner-text {
  color: $main-text-color;
  background-color: $light-background;
  border: $primary-border-width solid $dark-border;
}
```

Весь код из листингов 17.2 и 17.3 может находиться в одном файле. После его обработки препроцессором SCSS вы получаете код CSS, приведенный в листинге 17.4. Переменные заменяются значениями, определяющими действительное правило CSS.

Листинг 17.4. Разметка CSS, полученная в результате обработки листингов 17.2 и 17.3 препроцессором

```
.banner-text {
  color: #433e3f;
  background-color: #cecac4;
  border: 2px solid #7a7265; }
```

Импортирование и фрагменты

Пример с цветами можно немного развить и выделить палитру в отдельный файл. Если предположить, что весь код из листинга 17.2 хранится в файле с именем `_color-palette.scss`, вы можете включить его в основной файл SCSS (листинг 17.5). Имя файла палитры снабжается префиксом `_`, который является признаком фрагмента. Таким образом препроцессор узнает, что он не должен строить на основе этого файла CSS.

Листинг 17.5. Импортирование фрагмента SCSS

```
@import 'color-palette';

.banner-text {
  color: $main-text-color;
  background-color: $light-background;
  border: $primary-border-width solid $dark-border;
}
```

При таком подходе разработчик использует цветовую палитру вместе с дизайнером, а цветовая схема проходит контроль версии в системе управления исходным кодом. Мы можем продолжать разработку правил SCSS в то время, пока палитра еще не пришла к окончательному виду; эта палитра включается в исходный код везде, где требуется. При этом вы используете переменные, не беспокоясь о конкретных окончательных значениях.

Наследование

Если вы используете набор правил, основанный на базовом множестве стилей, вы можете воспользоваться средствами наследования SCSS. Как показано на рис. 17.6, для извлечения свойств из другого правила используется ключевое слово `@extend`.

Листинг 17.6. Использование `@extend` для наследования от другого стиливого правила

```
.simple-label {
  border: 1px solid #ccc;
  padding: 10px;
  color: #334;
}

.victory {
  @extend .simple-label;
  border-color: blue;
}

.defeat {
  @extend .simple-label;
  border-color: red;
}
```

SCSS берет код и объединяет правила в вывод, напоминающий листинг 17.7. Может показаться, что ничего особенного мы не достигли, но если объединить наследование с переменными и импортированием других файлов SCSS, возможности этого механизма значительно расширяются.

Листинг 17.7. Результат наследования от ранее определенного правила

```
.simple-label, .victory, .defeat {
  border: 1px solid #ccc;
  padding: 10px;
  color: #334; }

.victory {
  border-color: blue; }

.defeat {
  border-color: red; }
```

Вложение

Вложенные конструкции помогают упростить разметку CSS и немного приблизиться к структуре HTML. HTML и CSS не всегда хорошо совмещаются, но с вложением этот разрыв удастся немного сократить. Допустим, имеется структура HTML следующего вида:

```
<div>
  <p>Hello <span>world</span>!</p>
</div>
```

С вложением внутренний элемент `span` можно оформить следующим образом:

```
div {
  p {
    color: $main-text-color;
    span {
      font-weight: bold;
    }
  }
}
```

Вложение также помогает избежать постоянного повторения пространств имен CSS. В листинге 17.8 пространство имен `font` объявляется всего один раз, после чего задаются внутренние значения пространства имен.

Листинг 17.8. Сокращенная запись пространства имен в SCSS

```
div {
  p {
    color: $main-text-color;
    span {
```

```
        font: {
            weight: bold;
            size: 1.25em;
            style: italic;
        }
    }
}
```

После обработки препроцессором будет получен результат, приведенный в листинге 17.9. SCSS правильно упаковывает свойства пространства имен в синтаксис CSS и правильно генерирует два соответствующих правила.

Листинг 17.9. Стиливые правила, генерируемые на основе листинга 17.8

```
div p {
    color: #433e3f; }
div p span {
    font-weight: bold;
    font-size: 1.25em;
    font-style: italic; }
```

Учтите, что сложные вложения могут привести к проблемам с чрезмерным уточнением CSS. В случае листинга 17.9 на базе исходной разметки SCSS генерируются два правила, но дальнейшее вложение и объявления стилей на каких-либо уровнях селекторов могут создать проблемы. Если вам приходится создавать несколько вложенных блоков правил — скорее всего, с вашей системой стилей что-то не так. Вложение — отличный способ описания конкретных элементов, но при бездумном использовании оно может обернуться десятками непредвиденных (и скорее всего, ненужных) правил. Часто это происходит, когда разработчики не могут правильно изолировать элемент. Обычно вместо глубокого вложения существует более простой и удобный в сопровождении способ — например, пересмотр структуры HTML, добавление автономного класса или применение наследования вместо вложения.

Функции SASS Script

В настоящее время существует более 100 разных функций, которые могут использоваться для построения таблиц стилей. Идея использования функции на первый взгляд выглядит немного абсурдно (по крайней мере, нам так показалось), но потом ее преимущества постепенно начинают проявляться — например, при работе с цветами.

В листинге 17.10 мы возвращаемся к коду из листинга 17.2, в котором мы определяли цвета, но на этот раз для вычисления цвета фона используется одна из десятков цветовых функций.

Листинг 17.10. Использование встроенных функций SASS

```
$dark-border:      #7a7265;
$light-background: lighten($dark-border,35%);
$main-text-color:  #433e3f;
$primary-border-width: 2px;
```

Здесь к значению `$dark-border` применяется функция `lighten()` для получения более светлого оттенка цвета. В SASS Script имеются функции для работы с цветами, манипуляций со строками, выполнения вычислений, определения списков и карт (ассоциативных массивов) и многих других целей. Полный список функций SASS Script доступен по адресу <http://sass-lang.com/documentation/Sass/Script/Functions.html>.

Если вы следили за объяснениями и экспериментировали с основными возможностями, вероятно, вам уже хочется увидеть, как ваш код SCSS превращается в CSS. В следующем разделе речь пойдет о рабочих процессах и некоторых инструментах, но сначала мы рассмотрим еще одну полезную возможность, о которой нельзя не упомянуть, — примеси.

Создание примесей

Примесью (*mixin*) во многих языках программирования называется класс, «смешивающий» два и более класса (в отличие от наследования одного класса от другого). В SCSS примеси используются для группировки атрибутов и значений для многократного использования в разных элементах. Примесь можно рассматривать как шаблонную функцию, которая получает параметры и возвращает шаблон, заполненный переданными значениями. В следующем примере определение примеси `label` формирует такой шаблон для генерации класса CSS на базе переданного цвета:

```
@mixin label($base-color){
  color: $base-color;
  background-color: lighten($base-color,25%);
  border: 1px solid darken($base-color, 10%);
}
```

После этого можно использовать примесь в определении класса и передать цвет генерируемого поля:

```
.red-label { @include label(red); }
```

Результат выглядит так:

```
.red-label {
  color: red;
  background-color: #ff8080;
  border: 1px solid #cc0000; }
```

Использование примесей не ограничивается телом класса CSS. Собственно, вся мощь примесей по-настоящему проявляется тогда, когда вы объединяете, допустим, примесь `thick-border`, примесь `oversize` и примесь `label` для слияния их свойств в одном классе.

Объединение примесей и директив

Преимущества SCSS начинают проявляться при объединении всех возможностей, которые были описаны ранее. Например, цвета можно сохранить в переменных, применить к ним функции, а затем определить примесь для простой генерации соответствующих классов. Если пойти еще дальше, открываются совсем уж интересные возможности.

Давайте переделаем пример с `label`, но для начала взглянем на листинг 17.11: в нем представлена одна из директив для работы со списками — директива `@each`. Эта директива получает список, или ассоциативный массив, и перебирает его содержимое, генерируя стилевые правила на основании элементов списка.

Листинг 17.11. Применение директивы для генерирования стилевых правил

```
@each $label-color in red, green, blue {  
  .#{$label-color}-label { @include label($label-color); }  
}
```

В листинге 17.11 продемонстрированы некоторые новые возможности, поэтому его стоит рассмотреть подробнее. `red`, `green` и `blue` — просто элементы списка. В данном случае они интерпретируются как цвета при компиляции SCSS, потому что они являются известными именами цветов CSS; с таким же успехом вместо них можно было использовать шестнадцатеричные значения. Когда вы хотите использовать имя элемента как строковый литерал, заключите его в конструкцию `#{..}`, как в `#{$label-color}`. Эта конструкция используется для построения имени генерируемого класса. Наконец, мы вызываем ранее определенную примесь и передаем одно из значений из переданного списка. Результат выглядит довольно впечатляюще, если учесть, что листинг 17.11 состоит всего из трех строк кода.

```
.red-label {  
  color: red;  
  background-color: #ff8080;  
  border: 1px solid #cc0000; }  
  
.green-label {  
  color: green;  
  background-color: #01ff01;  
  border: 1px solid #004d00; }  
  
.blue-label {
```



```
color: blue;
background-color: #8080ff;
border: 1px solid #0000cc; }
```

Рабочий процесс

Не существует идеальной интегрированной среды (IDE), которая бы понравилась каждому разработчику на планете. Естественно, в организации рабочих процессов тоже сформировались разные схемы. У отдельного разработчика даже могут существовать свои предпочтения по работе с определенными проектами или типами файлов.

Инструменты командной строки

Если все, что вам нужно, — это просто выполнение сборок из командной строки, вам понадобится соответствующий инструментарий для SASS. Сайт *sass-lang.org* рекомендует использовать Ruby с загрузкой gem-пакетов, однако разработчики .NET все более привыкают использовать Node.js и менеджер пакетов npm. В npm существует пакет **node-sass**, который также имеет интерфейс командной строки. Он написан на языке C и работает очень быстро. Чтобы установить **node-sass**, введите следующую команду:

```
npm install node-sass
```

Затем найдите в структуре проекта место, в котором вам будет удобно обращаться к компилируемому файлу SCSS, и каталог для размещения полученной разметки CSS. Запустите **node-sass** для сборки файла CSS, передайте имена входного и выходного файла. В следующем примере команда выполняется из корня каталога **src** в вымышленном проекте:

```
node-sass style/site.scss wwwroot/css/site.css
```

В результате будет сгенерирован файл CSS **src/wwwroot/css/site.css**.

Работа в Visual Studio Code

В 2015 году компания Microsoft выпустила первую версию Visual Studio Code (или VS Code) — редактора с открытым кодом, работающего на всех платформах. От полной версии Visual Studio он отличается компактностью, возможностями настройки, расширением через быстро развивающийся репозиторий плагинов и поддержкой многих языков и типов сценариев.

Многим разработчикам нравится работать в VS Code, потому что этот редактор не несет всю нагрузку полноценной интегрированной среды разработки. VS Code рекомендует плагины, полезные в текущем контексте, и знает расширения фай-

лов, которые ими используются чаще всего — включая SCSS. Разработчики могут параллельно просматривать SCSS и выводимую разметку CSS, чтобы получить наглядное представление о создаваемых ими стилевых правилах. Чтобы эта возможность работала, следует также запустить инструмент `node-sass`, но с добавлением необязательного параметра `-w`:

```
node-sass -w style/site.scss wwwroot/css/site.css
```

На рис. 17.3 показано, как выглядит типичный сеанс создания таблицы стилей. В левой части изображена консоль с программой `node-sass`, отслеживающей изменения. Как только в отслеживаемый код SCSS или в импортируемые им файлы вносятся какие-либо изменения, `node-sass` выполняет препроцессорную обработку и автоматически обновляет выходной файл CSS, а VS Code в свою очередь мгновенно обновляет файл в правой части.

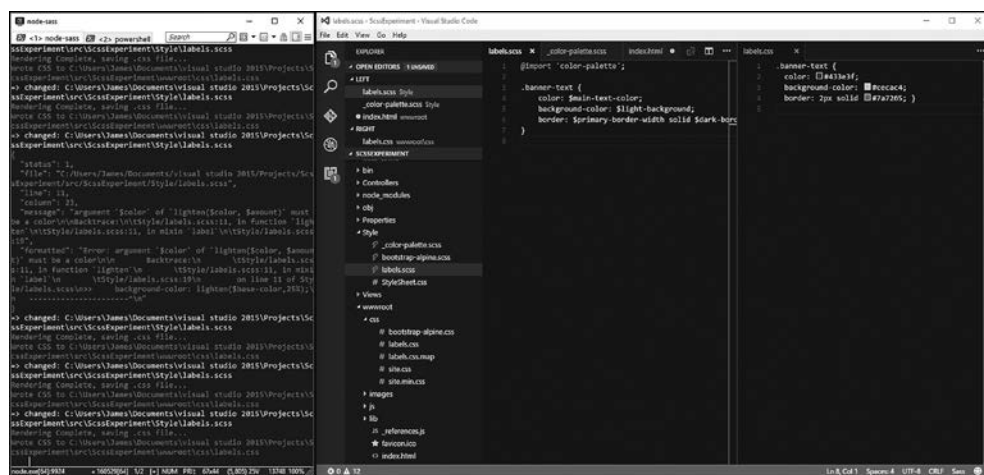


Рис. 17.3. Простой рабочий процесс с консолью и параллельными файлами в VS Code

Этот вариант организации рабочего процесса также хорошо подходит для изучения SCSS, потому что вы можете быстро вносить изменения, находить ошибки компиляции и немедленно видеть результат в редакторе.

Изменение задач сборки проекта

В команде Alpine было решено использовать файлы SCSS на билд-сервере и генерировать разметку CSS как артефакт сборки. Наряду с преимуществами, рассмотренными в этой главе, включение SCSS в конвейер сборки позволило команде Alpine генерировать одну таблицу стилей, включаемую в сборку. Таблица стилей, которая передается во внутреннюю среду для проверки, полностью совпадает с таблицей стилей, которая используется в рабочей версии приложения.

Чтобы включить файл SCSS в готовую задачу gulp, достаточно расширить шаблонное определение для файлов SASS и добавить звездочку для включения обоих разновидностей файлов, `.sass` и `.scss`.

```
sass: "style/**/*.*scss"
```

На всякий случай напомним, что обработка SASS, упаковка и сжатие кода рассматривались в главе 15 «Роль JavaScript». Не беспокойтесь, перед завершением этой главы вы увидите, как создается файл `bootstrap.scss`.

Использование сторонних фреймворков

Сторонняя библиотека или фреймворк CSS предоставляет простой задел для создания дизайна вашего сайта. Дизайнеры часто знают один или несколько фреймворков и используют их в качестве опорных точек в своем общении с разработчиками. Даже когда дизайнер решает двигаться в направлении, которое полностью отходит от того, что может предложить фреймворк, возможно, некоторые базовые конструкции все равно могут пригодиться.

Пара примеров, с которыми вы, возможно, уже знакомы — Bootstrap (getbootstrap.com) и Foundation (foundation.zurb.com). Эти библиотеки предоставляют средства нормализации макетов между разными браузерами и устройствами; они предоставляют гибкие сетки, которые хорошо адаптируются для экранов разного размера, а также предлагают язык проектирования и набор компонентов, с которыми вам будет легко создать профессионально выглядящее приложение. Часто в них имеются библиотеки JavaScript, которые добавляют расширенную функциональность и открывают возможность применения готовых решений, которые трудно реализовать на уровне CSS.

С другой стороны, для выбора стороннего фреймворка есть много причин, но только не наличие шаблонных дизайнов, с которыми многие сайты в интернете становятся похожими друг на друга. С учетом того, что вы знаете о CSS и SCSS, создать ваш собственный дизайн не так уж трудно.

Расширение фреймворка CSS

Если выяснится, что предлагаемая библиотекой функциональность по работе со стилями вас в целом устраивает, но вы не хотите ограничиваться базовыми возможностями, существуют два невероятно простых способа расширения фреймворка.

Способ первый: просто отредактируйте таблицу стилей. Да, это нормально! В недалеком прошлом наша команда работала над сайтом, и на всю работу от начала до конца у нас было около четырех недель. Клиент одобрил использование Bootstrap, но хотел обновить несколько цветов и сменить шрифт, чтобы он лучше соответ-

ствовал его логотипу. Мы взяли заранее построенную разметку CSS и выполнили поиск с заменой; на все изменения понадобилось меньше часа, а мы продолжили работу над кодом.

Второе прямолинейное решение — использование языка стилей фреймворка с включением дополнительных правил стиля, основанных на реализации. Недавно мы воспользовались этим решением для клиента, которому не нравилось, что значки уведомлений (badges) Bootstrap были только одного цвета. Используя схему, определенную в Bootstrap для кнопок, мы смогли создать для значков набор правил, которые соответствовали дизайну.

Эти решения «на скорую руку» удобны, но они не лишены недостатков. Если у фреймворка, с которого вы начинаете, выйдет обновление или исправление ошибки, обеспечить простое обновление вашего кода не удастся — вам придется тщательно следить за тем, какие изменения вы вносили. Работа с форматом CSS также может создавать свои проблемы: разрастающиеся файлы, пропущенные обновления цветов и коварные ошибки при отображении. Наконец, если вы используете лишь небольшое подмножество функциональности фреймворка, тем самым вы заставляете клиентов загружать всю библиотеку без каких-либо причин.

Как же сократить размер таблиц стилей, которые приходится загружать пользователю?

Настройка используемых элементов фреймворка CSS

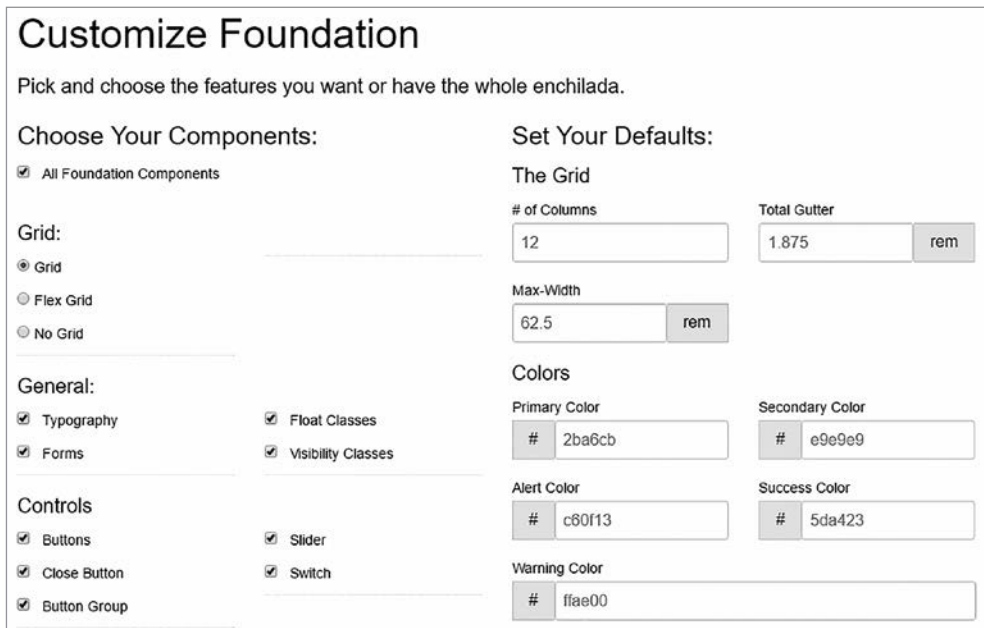
Как Bootstrap, так и Foundation предоставляют средства для выбора состава элементов, которые включаются в проект, и даже настройки реализации сетки, выбора цветов, шрифтов и других аспектов компонентов (рис. 17.4). После этого вы можете загрузить исходный код, обойтись без дополнительной настройки и получить ровно такой объем CSS, который вам нужен.

В случае Bootstrap даже строится файл JSON с описанием конфигурации вывода CSS. Вы можете перенести этот файл на сайт для внесения поэтапных изменений. С другой стороны, этот механизм не позволит использовать многие замечательные возможности SCSS, и вы не сможете создавать собственные компоненты на основании переменных, примесей и функций, предоставляемых такой библиотекой, как Bootstrap.

Применение фреймворков CSS для специализированных таблиц стилей

Именно ограничения, связанные с настройкой, повлияли на выбор команды Alpine. Вместо того чтобы просто загружать настроенную разметку CSS, было

решено использовать Bootstrap в качестве базы, а затем дополнить сайт собственными стилевыми правилами. Команда решила создать собственную версию Bootstrap из исходного кода, чтобы исключить неиспользуемые части и адаптировать систему для получения нужного результата.



Customize Foundation

Pick and choose the features you want or have the whole enchilada.

Choose Your Components:

☒ All Foundation Components

Grid:

☒ Grid
☐ Flex Grid
☐ No Grid

General:

☒ Typography
☒ Forms
☒ Float Classes
☒ Visibility Classes

Controls

☒ Buttons
☒ Close Button
☒ Button Group
☒ Slider
☒ Switch

Set Your Defaults:

The Grid

of Columns:

Max-Width:

Total Gutter:

Colors

Primary Color:

Secondary Color:

Alert Color:

Success Color:

Warning Color:

Рис. 17.4. Настройка Foundation на сайте Foundation

Этот процесс состоял из двух шагов. Сначала в проект был добавлен файл `bower.json`, напрямую поддерживаемый в Visual Studio 2015. Файл в конечном виде выглядел так:

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.4",
    "jquery": "2.2.0",
    "jquery-validation": "1.14.0",
    "jquery-validation-unobtrusive": "3.2.6"
  }
}
```

IDE распознает `bower.json` как файл, который объявляет зависимости и автоматически восстанавливает указанные пакеты в папку с именем `bower_components`

в каталоге проекта. Следующим шагом для команды Alpine стало простое копирование главного файла сборки Bootstrap SCSS в папку **Styles**, при этом файл был переименован в **bootstrap-alpine.scss**. Файл **bootstrap.scss** находится в каталоге **bower_components\bootstrap\scss** после того, как Visual Studio завершит восстановление ваших файлов.

С перемещением SCSS в каталог **Style** задача команды Alpine была практически выполнена; осталось только обновить относительное местонахождение файлов. Изначально модули SCSS, использованные для построения Bootstrap, находились в том же каталоге, поэтому директива импортирования:

```
@import "custom";
```

должна быть дополнена относительным путем:

```
@import "../bower_components/bootstrap/scss/custom";
```

Пути полностью зависят от структуры проекта. Задачи gulp были созданы ранее, необходимые зависимости были подключены (см. главу 1), что позволило строить итоговую разметку CSS «на лету». Разметка CSS также находится в папке **wwwroot/css** для удобства включения. Все это предоставляет отличную возможность для сокращения объема разметки CSS, передаваемой браузеру, за счет исключения ненужных компонентов и импортирования переменных или использования готовых примесей, предоставляемых Bootstrap.

Альтернативы для фреймворков CSS

Не всем нравится идея использования готовых фреймворков CSS. Высказываются немало обоснованных опасений: дело не только в том, что многие сайты начнут казаться похожими друг на друга, но и в том, что веб-разработчики никогда в полной мере не научатся писать CSS самостоятельно. Одна из самых серьезных претензий — загрузка всего фреймворка, иногда увеличивающая требования к загрузке на сотни килобайтов или даже мегабайты, только для использования меню или кнопки.

По этой причине появился ряд минималистичных фреймворков, ориентированных исключительно на такие аспекты, как работа со шрифтами и макетирование, но отказывающихся от цветовых схем и компонентов. И снова мы совершенно не пытаемся утверждать, что этот подход будет правильным для вашей команды. Найдите сбалансированное решение, которое хорошо работает в вашем проекте; исключите все лишнее, сгладьте все проблемные места и сделайте так, чтобы задачи настройки и сопровождения CSS в проекте для вашей команды решались по возможности легко и удобно.

Итоги

Создание CSS не должно сводиться к ручному воссозданию каждого правила, которое может понадобиться в вашем веб-приложении. Не существует идеального фреймворка CSS, равно как не существует идеального инструмента, препроцессора или рабочего процесса, который подходил бы для каждого проекта, — существует лишь то, что подойдет для вашей команды в конкретной ситуации. Когда вы начнете лучше ориентироваться в той области разработки, к которой относится CSS, вы создадите базовое решение, с которого можно будет легко переориентироваться на другие подходы. В руках разработчиков .NET находится постоянно расширяющийся набор фреймворков с качественной поддержкой и стартовыми инструментами, компенсирующими многие трудности разработки CSS, благодаря чему вы можете сосредоточиться на более важных аспектах своего приложения.

Оптимизация разметки CSS, передаваемой браузеру, улучшает впечатления конечного пользователя, но не является единственным способом улучшения субъективного быстродействия. В следующей главе мы покажем, как кэширование способно кардинально снизить нагрузку на сервер и при этом одновременно сократить время, в течение которого пользователь ожидает загрузки страницы.

18

Кэширование

Приближался день, когда приложение Alpine Ski House должно было заработать. Тим нанял парней из местного колледжа для тестирования, и все шло замечательно... по крайней мере сообщения в их отчетах сменились с «Происходит аварийное завершение Internet Explorer при записи в журнал» на сообщения «Дальтоникам будет трудно различать цвета кнопок». Балаш расставлял приоритеты задач, словно живой алгоритм быстрой сортировки. Команда лихорадочно трудилась в авральном режиме, поэтому видеть Тима на встрече было несколько неожиданно.

После того как все отчитались о ходе работы и текущих проблемах, Балаш сказал: «Вероятно, вы заметили, что здесь с нами Тим. Это необычно, но он хотел привлечь наше внимание к проблеме. Что ты хотел сказать нам, Тим?»

Тим сделал шаг вперед. «Спасибо, что позволили мне выступить. Я знаю, что обычно посторонним здесь говорить не положено. Дело в том, что мы отлично справлялись с нагрузкой, которую создавали наши тестировщики, но руководство носит с идеей проведения крупной рекламной акции через пару недель. Мы будем продавать билеты по таким низким ценам, что все просто попадают на пол. И я боюсь, что сайт упадет от наплыва клиентов. Что мы можем сделать, чтобы защититься от этого?»

Первым заговорил Честер. «Мы можем поднять численность нагрузочных тестов. Может быть, потратить время на настройку параметров. Нас это не спасет, но по крайней мере мы будем знать о потенциальных проблемах».

«Думаю, можно заняться оптимизацией запросов, потому что пара наших запросов создает основательную нагрузку на сервер базы данных. Индексов много не бывает!» — предположил Адриан.

«Почему бы не добавить кэширование? — спросила Кэндис. — Я что-то читала о том, что в ASP.NET MVC можно добавить кэширование вывода».

«И как это поможет? — спросил Тим. — Люди же все равно будут приходить на сайт?»

«Не обязательно, — ответила Кэндис. — Но даже если и будут, нагрузка на базу данных и веб-серверы будет снижена из-за кэширования промежуточных данных».

«Ладно, — ответил Тим. — Тогда проведем нагрузочное тестирование, скажем, для пятисот одновременно работающих пользователей, подчистим запросы и добавим кэширование».

Снижение нагрузки на веб-серверы, чтобы они могли быстро возвращать страницы, всегда является положительным фактором. Существует ряд приемов, направленных на перемещение нагрузки с веб-серверов или на сокращение количества операций, фактически выполняемых сервером. Все эти приемы обозначаются одним термином — «кэширование».

Если речь заходит об организации хранения данных на компьютере, о чем мы обычно думаем? Жесткий диск, память, USB-устройства, пятидюймовые дискеты... Ладно, про последний пункт уже почти никто не вспоминает, но есть немало мест для хранения данных, о которых мы обычно вообще не думаем.

Самое быстрое место для хранения данных на компьютере — регистры процессора. На микросхеме X86-64 размещены шестнадцать 64-разрядных регистров, а также целая коллекция регистров, добавленных как часть расширений процессора, таких как SIMD и x87. Кроме регистров, также процессор оснащен кэшами L1, L2 и L3.

При переходе от регистров к трем уровням кэширования объем памяти увеличивается, но сама память работает медленнее. Следующая по размеру память — оперативная память (ОЗУ) — все еще работает достаточно быстро. Наконец, жесткий диск предназначен для хранения больших объемов данных с медленным доступом. В табл. 18.1 приведена сводка приблизительного времени обращения к различным видам памяти.

Таблица 18.1. Приблизительное время обращения к различным видам памяти

Местонахождение	Время обращения
Кэш L1	0,5 наносекунды
Кэш L2	7 наносекунд
Оперативная память	100 наносекунд
Жесткий диск SSD	150 микросекунд

ПРИМЕЧАНИЕ

Если вы захотите больше узнать об архитектуре современных процессоров, компания Intel опубликовала ряд очень интересных документов, включая руководство для разработчиков ПО для архитектур Intel 64 и IA-32, по адресу <http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf>. Документ чрезвычайно интересный, хотя довольно длинный (4670 страниц). Радуйтесь, что вы работаете на уровне абстракции, на котором все это знать не обязательно!

Разные уровни кэширования на современных процессорах служат одной цели — они ускоряют доступ к информации. Предполагается, что после обращения к какой-либо информации вы с большой вероятностью захотите обратиться к ней снова в ближайшем будущем. Когда информация запрашивается во второй раз, она может уже храниться в кэше, и тогда нам не придется нести затраты, связанные с обращением к оперативной памяти или того хуже — к диску.

Кэширование на веб-сервере решает ту же задачу. Во многих случаях обновления данных на сайте происходят относительно редко. Лучшим примером служит страница входа на сайт: чаще всего страница входа получает наибольшее количество обращений, но при этом содержит в основном статические данные. Например, домашняя страница сайта <http://www.microsoft.com> для разных пользователей выглядит совершенно одинаково. Обновления могут происходить несколько раз в день, при появлении новых сообщений или повышенном внимании к какому-либо контенту. Вместо того чтобы нести затраты по выполнению кода в контроллере, выборке данных из базы данных, а затем отображать контент в представлении, возможно, мы можем сохранить вывод страницы или использовать уже созданные ранее ее части.

Заголовки управления кэшированием

Протокол HTTP строился с учетом возможности кэширования. В спецификации определены команды, которые наверняка будут знакомы всем, кто работал с хорошо спроектированными REST-совместимыми API. Команда GET, которая используется каждый раз, когда вы вводите URL в браузере и нажимаете клавишу Enter, попросту загружает ресурс. Она не должна изменять данные или состояние на сервере. Запросы GET обычно могут безопасно кэшироваться прокси-серверами между клиентом и сервером. Такие команды, как POST и PATCH, изменяют состояние на сервере, а следовательно, не являются безопасными для кэширования, потому что они должны пройти к серверу для обновления данных (рис. 18.1).

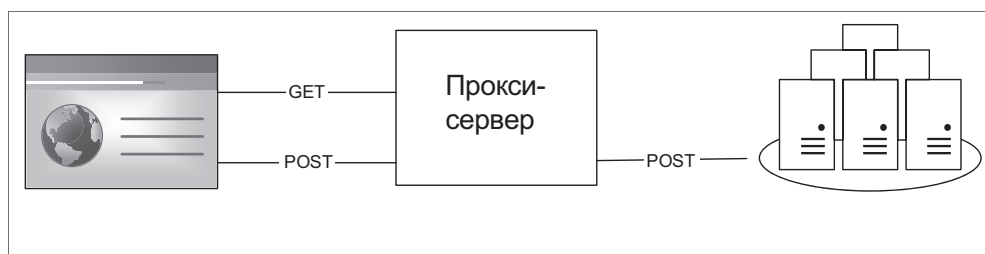


Рис. 18.1. Прокси-сервер перехватывает запросы GET, но пропускает запросы POST на сервер

Иногда бывает нежелательно, чтобы запросы GET кэшировались прокси-серверами или клиентами. Возможно, данные быстро изменяются, или страница задействована в цикле «список — редактирование — список», и пользователи хотят немедленно видеть свои данные в обновленном виде. В других случаях нас устраивает то, что прокси-сервер или клиент кэширует страницы в течение какого-то времени. Для управления кэшированием информации на прокси-серверах и клиентских кэшах используются заголовки для управления кэшированием.

КЭШИРОВАНИЕ ВЫВОДА

В предыдущих версиях ASP.NET MVC существовала возможность добавления в действия директив кэширования, которые приказывали фреймворку сохранить вывод и предоставлять его в будущем вместо выполнения действия. Код выглядел примерно так:

```
[OutputCache(5)]
public ActionResult Index()
{
    // Затратная операция
}
```

К сожалению, на момент написания книги такая разновидность кэширования в ASP.NET Core MVC еще не поддерживается, но она входит в список функций, заявленных для версии 1.1 фреймворка. Она будет реализована в форме промежуточного ПО.

В действие контроллера могут добавляться полезные атрибуты, обеспечивающие добавление заголовков для управления кэшированием в ответах. В отличие от кэширования вывода, упоминавшегося выше, это кэширование выполняется не на сервере, а на клиентских машинах или на промежуточном прокси-сервере. Например, чтобы кэширование страницы заведомо не использовалось, применяйте следующую конструкцию:

```
[ResponseCache(Location = ResponseCacheLocation.None, NoStore = true)]
public IActionResult Index()
{
    // Операция, которая не должна кэшироваться
}
```

Заголовку `Cache-Control` присваивается значение `no-store;no-cache`, которое запрещает сохранение контента на прокси-сервере. Как ни странно, просто указать `no-cache` недостаточно, потому что это значение всего лишь приказывает прокси-серверу сверяться с сервером HTTP перед выдачей контента. Добавление `no-store` гарантирует, что прокси-сервер действительно будет передавать запросы серверу HTTP.

Чтобы контент хранился на прокси-сервере, необходимо указать другие заголовки. Вы можете управлять двумя аспектами кэширования: его продолжительностью и доступностью (для отдельного пользователя или для всех). Продолжительность задается так:

```
[ResponseCache(Duration = 60)]
public IActionResult Index()
{
    // Операция, которая должна кэшироваться 60 секунд
}
```

Продолжительность задается в секундах; к сожалению, она определяется целым числом вместо значения `TimeStamp`, которое было бы более понятным. Даже группа ASP.NET иногда склонна увлекаться примитивами. Атрибут `ResponseCache` устанавливает в заголовке `Cache-Control` поле с именем `max-age`. Атрибут `Location` может принимать значения `Any` и `Client`, соответствующие `public` и `private` в заголовке HTTP. Значение `public` означает, что контент может кэшироваться прокси-серверами на пути к клиентской машине, а значение `private` означает, что контент может сохраняться только браузером на машине клиента. Для любого общедоступного контента (например, домашней страницы) указывайте `Any`, а для всех страниц, относящихся к конкретному пользователю (например, страниц профиля), указывайте `private`. Общедоступные страницы могут предоставляться всем пользователям, запросы которых проходят через прокси-сервер. Однако следует заметить, что выбор значения `private` не повышает безопасность, потому что он всего лишь требует, чтобы контент не сохранялся на общедоступных прокси-серверах.

```
[ResponseCache(Duration = 60, Location = ResponseCacheLocation.Client)]
public IActionResult Index()
{
    // Операция кэшируется 60 секунд и относится к конкретному пользователю
}
```

Необходимость задавать заголовки для управления кэшированием при каждом действии слегка раздражает. Если вдруг потребуется поднять продолжительность кэширования с 60 секунд до 90, совершенно не хочется вносить изменения во всех действиях, для которых был установлен атрибут `ResponseCache`. Для решения этой проблемы можно воспользоваться профилями кэширования. При настройке сервисов в `Startup.cs` профили добавляются следующим образом:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(options =>
    {
        options.CacheProfiles.Add("Never",
            new CacheProfile())
    })
}
```

```
        {
            Location = ResponseCacheLocation.None,
            NoStore = true
        });
    });
    options.CacheProfiles.Add("Normal",
        new CacheProfile()
        {
            Duration=90;
            Location = ResponseCacheLocation.Client;
        });
}
```

Пример применения профиля:

```
[ResponseCache(CacheProfileName = "Normal")]
public IActionResult Index()
{
    // Операция, которая должна кэшироваться с настройками Normal
}
```

Заголовки для управления кэшем полезны, но они зависят от того, кэширует ли клиент информацию, а мы этого гарантировать не можем. Кроме того, точность заголовков невелика: они применяются к ресурсу в целом, а не к отдельным частям страницы. Если нужно кэшировать только отдельные части страницы, заголовки вам в этом не помогут. Нужен другой подход.

Использование Data-Cache

Результаты затратных операций на сервере можно сохранять в кэше, чтобы эти операции не приходилось повторять заново. Примером такого рода может послужить запрос к базе данных для получения набора записей. Если результаты запроса изменяются не так часто, они могут кэшироваться в течение нескольких секунд и даже нескольких минут. Впрочем, при этом важно учесть возможные последствия от получения пользователем устаревшей информации. Такие данные, как список отелей в Ванкувере, вряд ли будут часто изменяться и могут кэшироваться в течение относительно длительного времени, а биржевые котировки могут изменяться каждую секунду, и поэтому не должны кэшироваться.

Кэширование в ASP.NET Core существует в двух вариантах: кэширование в памяти и распределенное кэширование. В первом случае данные сохраняются в памяти на компьютере, а при распределенном кэшировании данные помещаются в некое хранилище данных, внешнее по отношению к серверу HTTP. Начнем с первого варианта.

Кэширование памяти

Для начала необходимо включить кэширование памяти для проекта, добавив пакет `nuget` в файл `project.json`:

```
"dependencies": {
  ...
  "Microsoft.Extensions.Caching.Memory": "1.0.0"
}
```

Затем этот сервис включается в регистрацию сервисов в файле `Startup.cs`:

```
public void ConfigureServices(IServiceCollection services)
{
  ...
  services.AddMemoryCache();
  ...
}
```

В контейнер добавляется сервис, реализующий `IMemoryCache`. Он может использоваться везде, где может использоваться контейнер внедрения зависимостей, как в следующем примере с `SkiCardController`:

```
public SkiCardController(SkiCardContext skiCardContext,
                        UserManager<ApplicationUser> userManager,
                        IAuthorizationService authorizationService,
                        IBlobFileUploadService uploadService,
                        IMemoryCache memoryCache,
                        ILogger<SkiCardController> logger)
{
  _memoryCache = memoryCache;
```

Стандартная схема использования кэша памяти (или любого другого кэша) — проверка содержимого кэша и использование значения, если оно присутствует. Если же значение отсутствует в кэше, то выполняется операция, а ее результат сохраняется в кэше. `IMemoryCache` предоставляет операции `TryGetValue`, `CreateEntry` и `Remove` для работы с данными. Также существуют методы расширения, с которыми этот интерфейс становится еще удобнее. Например, если вы захотите использовать кэш в памяти для хранения значения в течение 1 минуты, это делается так:

```
SkiCard skiCard = null;
if(!_memoryCache.TryGetValue($"skicard:{viewModel.Id}", out skiCard))
{
  skiCard = await _skiCardContext.SkiCards
    .SingleOrDefaultAsync(s => s.Id == viewModel.Id);
  _memoryCache.Set($"skicard:{viewModel.Id}", skiCard,
    new MemoryCacheEntryOptions().SetAbsoluteExpiration(TimeSpan.
      FromMinutes(1)));
}
```

СОВЕТ

Важно продумать хорошую схему построения ключей кэширования, чтобы кэш не забивался неожиданными значениями. Один из возможных вариантов — «имя объекта:идентификатор» — например, `skiCard:{id}` или `user:{id}`.

Объект `skiCard` будет извлекаться из кэша или из используемой базы данных. Если значение присутствует в кэше, обращаться к базе данных не нужно. Срок действия записи в кэше истекает через 1 минуту после того, как она появилась в кэше.

Распределенный кэш

С одним веб-сервером кэширование данных в памяти работает быстро и эффективно. Но стоит вам перейти на конфигурацию с несколькими веб-серверами, и сразу появляется необходимость в централизованном хранилище данных. Было бы хорошо, если бы кэшированные данные на сервере А могли быть прочитаны сервером Б. Для этого необходимо воспользоваться распределенным кэшем. Кэш в памяти и распределенный кэш реализуют разные интерфейсы в ASP.NET Core, хотя они и делают приблизительно одно и то же. Существуют вполне реальные ситуации, в которых нужно выбирать, какие данные должны кэшироваться в памяти, а какие — в распределенном кэше, поэтому наличие разных интерфейсов оправданно.

Microsoft предоставляет две реализации распределенного кэша: для SQL Server и для Redis. Выбор того, какая реализация лучше подходит для вашего проекта, в основном сводится к тому, с чем ваша группа чувствует себя более уверенно. Пожалуй, реализация Redis обладает большей эффективностью и простотой в настройке. Redis идеально подходит для кэширования, потому что представляет собой хранилище пар «ключ — значение». Впрочем, многие организации еще не привыкли к изменениям в окружающем мире и предпочитают использовать более знакомые технологии — в данном случае SQL Server. По крайней мере в течение десятилетия SQL Server решал все задачи хранения данных в технологическом стеке .NET. Сейчас ситуация изменяется, и мы начинаем понимать, что требования к хранению данных различны, а значит, технологии тоже одинаковыми быть не могут.

Какой бы тип хранения вы ни выбрали, распределенный кэш предоставляет унифицированный интерфейс, так что сменить схему кэширования «на лету» будет несложно. Как и во всех аспектах ASP.NET Core, реализация распространяется с открытым кодом и имеет модульную структуру, так что вы можете создать собственную схему хранения данных. Например, если ваша компания серьезно занимается Apache Casandra, вы сможете относительно легко организовать кэширование на базе этой технологии.

Снова все начинается с подключения соответствующей реализации кэша в `project.json`. Подключить можно реализацию Redis или SQL Server:

```
"Microsoft.Extensions.Caching.Redis": "1.0.0",  
"Microsoft.Extensions.Caching.SqlServer": "1.0.0"
```

Затем необходимо настроить и зарегистрировать реализацию в контейнере. Для Redis конфигурация выглядит так:

```
services.AddDistributedRedisCache(options =>  
{  
    options.Configuration = Configuration.GetValue("redis.host");  
    options.InstanceName = Configuration.GetValue("redis.instance");  
});
```

Для SQL Server конфигурация выглядит так:

```
services.AddDistributedSqlServerCache(options =>  
{  
    options.ConnectionString = Configuration.GetConnectionString("cache");  
    options.SchemaName = Configuration.GetValue("cache.schemaName");  
    options.TableName = Configuration.GetValue("cache.tableName");  
});
```

В обоих случаях мы читаем значения из файла конфигурации для настройки подключения.

ПРИМЕЧАНИЕ

Redis относится к бессхемным базам данных, и это довольно удобно, потому что вы можете добавить в базу данных практически любую информацию. SQL Server предъявляет более жесткие требования к схеме, поэтому схему необходимо строить заранее. Для этого можно установить пакет `Microsoft.Extensions.Caching.SqlConfig.Tools` и включить его в секцию инструментов файла `project.json`.

```
"tools": {  
    "Microsoft.Extensions.Caching.SqlConfig.Tools": "1.0.0-*",  
}
```

Это позволит вам выполнить команду `dotnet sql-cache create`, которая создаст таблицы кэша за вас.

После настройки распределенного кэша вы сможете работать через контейнер DI с интерфейсом `IDistributedCache`, похожим на `IMemoryCache`. Он используется примерно так:

```
var cachedSkiCard = await _distributedCache.GetAsync($"skicard:{viewModel.Id}");  
if(cachedSkiCard != null)  
{
```



```
        skiCard = Newtonsoft.Json.JsonConvert.DeserializeObject<SkiCard>
            (cachedSkiCard);
    }
    else
    {
        skiCard = await _skiCardContext.SkiCards
            .SingleOrDefaultAsync(s => s.Id == viewModel.Id);
        await _distributedCache.SetStringAsync($"skicard:{viewModel.Id}",
            Newtonsoft.Json.JsonConvert.SerializeObject(skiCard), new
            DistributedCacheEntryOptions()
            .SetAbsoluteExpiration(TimeSpan.FromMinutes(1)));
    }
}
```

В этом коде следует обратить внимание на пару моментов. Во-первых, методы кэша являются асинхронными, поэтому они используются с ключевым словом `await`. Также существуют синхронизированные версии, но с точки зрения быстродействия кода асинхронные версии предпочтительны. Во-вторых, распределенный кэш работает только с массивами байтов и строками, и сложные объекты (такие, как `skiCard`) приходится сериализовать. В данном примере для выполнения сериализации и десериализации используется библиотека `Newtonsoft.Json`, но вы можете выбрать другие форматы.

Ограничение размера кэша

На первый взгляд кэширование большого объема данных выглядит очень заманчиво, но некоторые проекты заходят в кэшировании слишком далеко. Некоторые разработчики считают, что информация в раскрывающихся списках обновляется слишком медленно; так почему бы не кэшировать ее на стороне клиента, чтобы за ней не приходилось обращаться на сервер? Как насчет поиска отдельного значения в базе данных? Эти данные изменяются нечасто, поэтому их можно кэшировать. Однако понять, когда именно содержимое кэша должно становиться недействительным, непросто. Если данные в раскрывающемся списке действительно изменятся, хранение их на каждой клиентской машине создаст проблемы с обновлением распределенного кэша.

Кэширование должно выдерживать баланс между простотой сопровождения и последствиями от работы пользователя с устаревшими данными. Также в некоторых ситуациях обращение к базе данных не уступает по скорости кэшированию, особенно в конфигурациях с распределенным кэшем. Прежде чем принимать решение о выделении значительных ресурсов для кэширования, поэкспериментируйте и убедитесь в том, что кэширование действительно помогает. При добавлении кэша используйте подход, основанный на данных.

Итоги

ASP.NET MVC предоставляет несколько методов кэширования данных: с кэшем на прокси-сервере и на стороне браузера клиента, а также на самом сервере. Кэш памяти — простейшая из имеющихся реализаций кэша, хорошо подходящая для односерверных конфигураций или для данных, которые не обязательно распределять между серверами. Распределенный кэш использует для хранения данных внешние средства — например, Redis и SQL Server. В результате один сервер может сохранить данные в кэше, а другой сервер прочитает эти данные. Кэширование обычно улучшает быстродействие и снимает нагрузку с плохо масштабируемых ресурсов (например, реляционных баз данных); тем не менее важно провести хронометраж и убедиться в том, что этот механизм помогает, а не вредит вашему приложению.

В следующей главе рассматриваются компоненты, которые можно использовать повторно, а также возможности их применения для ускорения работы сайта и обеспечения его целостности.

ЧАСТЬ IV

Спринт четвертый: Финишная прямая

Глава 19. Многоразовый код.....	362
Глава 20. Тестирование.....	377
Глава 21. Расширение фреймворка.....	399
Глава 22. Интернационализация.....	416
Глава 23. Рефакторинг и повышение качества кода	429
Глава 24. Организация кода	450

Даниэль была потрясена до глубины души. Она только что досмотрела совершенно жуткую серию «Секретных материалов» и весь день сидела в напряжении. Сезон 4, серия 2. Каждый раз, вспоминая о ней, она судорожно выдыхала и качала головой. Если ей придется и дальше спать с включенным светом, надо будет купить энергосберегающих светодиодных лампочек. К тому же есть особенно яркие лампочки, которые осветят всю комнату, и ничто не сможет подкрасться к ней тайком. Она как раз оплачивала экспресс-доставку со своего телефона, когда зашел Балаш.

В руках он держал большую коробку. «Я купил футболки на всю команду!» — радостно заявил он. Балаш вытащил из коробки одну из белых футболок. Спереди ее украшал логотип Alpine Ski House. Даниэль была немного разочарована; она бы предпочла что-нибудь связанное с проектом. Балаш перевернул футболку: на спине была карикатура, изображающая пучок петрушки, с одобрительно поднятыми вверх большими пальцами. Завершала это великолепие надпись: «Проект Parsley».

«Я кое-что знаю о хороших шмотках, — сказал Честер, — футболки просто зачетные».

Даниэль не знала, хорошо это или плохо, но футболки ей понравились. Похоже, другие участники были согласны — когда Балаш раздавал футболки, в комнате стояло одобрительное бормотание.

«Так, хорошо, — перебил Балаш, — пора переходить к ретроспективе. Правила вы уже знаете, берите маркеры — посмотрим, как у нас шли дела на этой неделе».

Начать	Прекратить	Продолжать
Реорганизовать приложение	Выпускать версии с таким количеством ошибок	Использовать Azure для хостинга
Подготовить больше автоматизированных тестов	Включать столько кода в контроллеры, в некоторых из них по 300 строк	Использовать внедрение зависимостей
Повторно использовать код, встречающийся в нескольких местах		Делать всякие классные штуки на JavaScript
		Носить классные футболки

«Мне не нравится, что тестирование снова упоминается в первом столбце. Разве мы с этим не разобрались в прошлый раз?» — спросил Балаш.

«Не совсем, — ответил Марк-2. — Мы провели обеденный семинар, и я думаю, что большая часть команды в курсе дел, но отмашку еще не дали».

Метод разработки через тестирование произвел большое впечатление на Даниэль. Она была уверена, что он повышает качество кода, но до сдачи проекта оставалась всего неделя. На столь поздней стадии проекта серьезные изменения были немыслимы. Марк-2 объяснил, что программисты по-разному относятся к разработке через тестирование, и Даниэль была уверена, что она ближе к середине спектра мнений — и уж во всяком случае не принадлежит к кругу «фанатов», как Марк-2 описывал некоторых людей.

«Я понимаю, — сказал Балаш. — На такой поздней стадии проекта до всего руки не доходят. Ребята, я вами горжусь! Все основные функциональные блоки готовы. Страницы загружаются. Они хорошо смотрятся, и, похоже, мы даже разобрались с интеграцией кредитных карт. Знаю, мы уже давно вкалываем в авральном режиме, но у нас осталась всего неделя, чтобы произвести впечатление на начальство. И мы справимся! Задачи следующего спринта — заполнить пробелы и выслушать тестировщиков».

Тим вступил в разговор: «Я посмотрел цифры — вчера у нас было 48 развертываний в тестовой среде. Глазам своим не верю, это потрясающе! Тестировщики говорят, что от обнаружения ошибки до ее исправления обычно проходит всего пара часов. Так что носите футболки с гордостью».

Дела у команды шли хорошо, и Даниэль была счастлива. Успехи почти стерли из памяти ту жуткую серию о том, как... стоп, она потрянула головой — она должна с этим справиться.

«Ладно, похвалили себя — и будет, — сказал Балаш. — Работа еще не закончена. Я вижу претензию к длине контроллеров — кто это написал?»

«Это я, — сказал Марк-2. — Некоторые из этих монстров содержат более 300 строк. У нас будут проблемы с сопровождением кода, и не будем забывать о конфликтах слияния, когда над файлом работают сразу несколько людей. Предлагаю разделить на несколько частей».

«Мне это не нравится, — прервала Даниэль. — То есть я тоже терпеть не могу длинные файлы, но работа уже зашла слишком далеко, чтобы менять такие вещи, как URL. Я против перемещения действий в другие контроллеры».

«Об этом и речи не идет, — ответил Марк-2. — Должны быть какие-нибудь средства для вынесения кода из контроллера без фактического изменения маршрутов. Обновить таблицу маршрутизации или вроде того».

«Да, или вроде того», — согласилась Даниэль.

«Хорошо, — сказал Балаш. — Не хочу задерживать вас зазря, идите и исправьте все, что нужно исправить. Мы на финишной прямой!»

19

Многоразовый код

Кодовая база продолжала расти, когда Даниэль вдруг услышала, как Марк-2 стучит по клавишам своей механической клавиатуры за перегородкой. Клавиатура была шумная, но Даниэль не возражала: она работала в наушниках и целый день слушала Ену. Однако клацанье было громче обычного, поэтому она заглянула за перегородку.

«Марк, что случилось?»

«Привет, Даниэль. С клавиатурой проблемы. Клавиши Ctrl, C и V стали работать хуже. Приходится лупить, чтобы они сработали».

«Я слышала, что механические переключатели рассчитаны на 50–60 миллионов нажатий. Не может быть, чтобы ты их все израсходовал», — пошутила Даниэль.

Однако Марк-2 не улыбнулся. «Да, но сейчас я просто копирую большой объем кода между разными местами сайта. С серверной частью все нормально, я могу использовать классы, но в интерфейсной части только и остается, что нажимать Ctrl+C, Ctrl+V».

«Ты хочешь сказать, — спросила Даниэль, — что с представлениями Razor нельзя использовать наследование?»

«В общем-то нет. Как ты это себе представляешь? Переопределить часть страницы? Создать полиморфные представления? Довольно странно».

«Я понимаю, что это было бы странно, но наследование — не единственная форма повторного использования кода. Пожалуй, композиция лучше подходит для кода интерфейсной части. Позволь, я покажу тебе, как избавиться от лишнего копирования и вставки».

«Было бы здорово. Не думаю, что Тим раскошелится на новую клавиатуру для меня».

Всем нам попадалось сокращение DRY, от «Don't Repeat Yourself», то есть «не повторяйтесь». По поводу этого принципа в разработке часто идут баталии, потому

что предотвращение дублирования кода и логики приложения может быть непростым делом. Дублирование кода усложняет работу по сопровождению кода, потому что изменения приходится вносить сразу в нескольких местах; что еще хуже, возникает риск внесения ошибок из-за того, что изменения будут пропущены в каких-то экземплярах дублируемого кода. К сожалению, борьба с дублированием иногда приводит к столь же вредному возрастанию сложности кода. Нет смысла обеспечивать стопроцентное повторное использование кода, если полученная реализация будет слишком сложной и в ней не разберется ни один разработчик.

К счастью, в ASP.NET Core MVC включены полезные функции, которые позволяют инкапсулировать компоненты пользовательского интерфейса так, что при правильной реализации код получается понятным и простым для многоразового использования. В этой главе будут рассмотрены тег-хелперы, компоненты представлений и частичные представления.

Тег-хелперы

Тег-хелперы, поддерживаемые механизмом представлений Razor, — превосходный способ инкапсуляции небольших фрагментов логики представления и обеспечения последовательной генерации разметки HTML. В главе 11 «Представления Razor» было показано, как использовать тег-хелперы для чрезвычайно четкого и компактного описания форм. Но прежде чем рассматривать некоторые встроенные тег-хелперы и разбираться с тем, как создавать их самостоятельно, необходимо разобраться, как же они устроены.

Строение тег-хелперов

Встроенные тег-хелперы полезны, но самое интересное начинается, когда вы переходите к созданию собственных тег-хелперов. Но прежде чем их создавать, следует понять, из каких частей состоит тег-хелпер. Для примера рассмотрим тег-хелпер `input` на рис. 19.1.

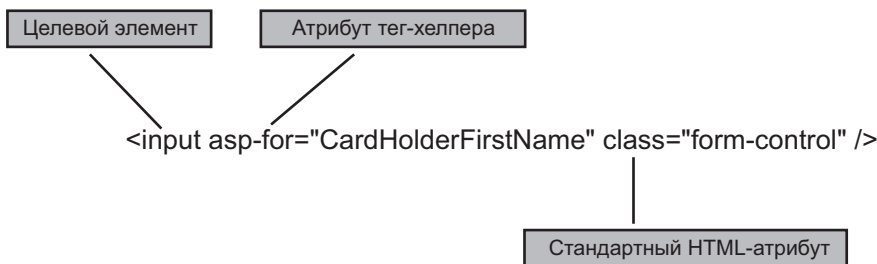


Рис. 19.1. Пример использования тег-хелпера

Тег-хелпер предназначен для конкретного элемента HTML, называемого целевым элементом. В случае тег-хелпера `input` целевым является элемент `input`. Тем не менее это не означает, что тег-хелпер `input` выполняется для каждого элемента `input` в приложении — он применяется только к элементам `input`, которые также обладают атрибутом `asp-for`. Атрибут `asp-for` является примером атрибута тег-хелпера. Атрибуты используются для передачи параметров конкретному тег-хелперу. Комбинация из целевого элемента и атрибута заставляет Razor интерпретировать элемент как тег-хелпер. У некоторых тег-хелперов целевыми являются существующие элементы HTML, у других — нестандартные элементы (как, например, у тег-хелпера `cache`).

Атрибуты тег-хелпера определяются для конкретного типа .NET — например, `string`, `int` или `TimeSpan`. Атрибут `asp-for` является специальным атрибутом типа `ModelExpression`, который сообщает Razor, что атрибут должен получать выражение, ассоциируемое с текущей моделью.

Тег-хелпер также может содержать любые другие действительные атрибуты HTML, применяемые к выходной разметке HTML. Некоторые тег-хелперы (такие, как `cache`) могут содержать дочерний контент. В зависимости от тег-хелпера дочерний контент может быть каким-то образом изменен или упакован в разметку HTML, сгенерированную тег-хелпером.

Тег-хелперы `environment`, `link` и `script`

Тег-хелпер `Environment` предоставляет простой способ визуализации разных фрагментов HTML в зависимости от текущего окружения. Например, можно использовать сокращенные файлы CSS в среде промежуточного развертывания и в рабочей среде, но при этом использовать полные версии в среде разработки. Текущие данные среды читаются из переменной `ASPNET_ENVIRONMENT`, а если значение этой переменной не задано, ASP.NET считает, что оно равно `Production`.

```
<environment names="Development">
  <link rel="stylesheet" href="~/css/site1.css" />
  <link rel="stylesheet" href="~/css/site2.css" />
</environment>
<environment names="Staging,Production">
  <link rel="stylesheet" href="~/css/site.min.css"/>
</environment>
```

Клиенту отправляется только содержимое тег-хелпера `environment`, а сам тег `environment` в вывод HTML не включается. Чаще всего тег-хелпер `environment` используется с тег-хелперами `link` и `script`. Тег-хелперы `link` и `script` предоставляют параметры для подстановки, обращений к ресурсам из инфраструктур CDN и отключения кэширования.

Файловые шаблоны

Тег-хелперы `link` и `script` предоставляют средства для определения файлов с использованием шаблонов. Например, если вы хотите определить все файлы с расширением `.js` в папке `scripts`, укажите шаблон `~/scripts/**/*.js` в значении атрибута `asp-src-include` тег-хелпера `script`:

```
<script asp-src-include="~/scripts/**/*.js" ></script>
```

Тег-хелпер `script` генерирует отдельный тег `script` для каждого файла, соответствующего заданному шаблону. Вы также можете использовать дополнительный атрибут `asp-src-exclude` для исключения файлов. Тег-хелпер `link` работает аналогичным образом: он тоже предоставляет возможность задания файловых шаблонов.

Запрет кэширования

Запрет кэширования (*cache busting*) — это процесс присоединения некоторой разновидности хеша версии файла к именам файлов ресурсов (например, файлов JavaScript и CSS). Преимущество такого решения заключается в том, что вы можете приказать браузеру кэшировать такие файлы в течение неограниченного времени, не беспокоясь о том, что клиент получит старую версию при изменении файла. Поскольку имя ресурса меняется при изменении его содержимого, загружаться всегда будут обновленные файлы. Чтобы активизировать запрет кэширования, достаточно включить атрибут `asp-append-version="true"` для тег-хелперов `link` и `script`.

```
<link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true"/>
```

Во время выполнения тег-хелпер генерирует хеш версии и присоединяет его в виде параметра запроса к URL-адресу файла.

```
<link rel="stylesheet" href="/css/site.min.css?v=UdxKHVNJ42vb1EsG909uURADfEE3j1E3Dg  
wL6NiDFOe" />
```

Запрет кэширования также может использоваться с графическими ресурсами, для чего к тегам `img` добавляется атрибут `asp-append-version="true"`.

CDN и резервирование

Один из распространенных методов оптимизации основан на подключении ссылок на популярные фреймворки из инфраструктур CDN (Content Delivery Network) для сокращения сетевой нагрузки на ваш сервер и, возможно, с улучшением быстродействия для пользователя. Для популярных фреймворков, таких как

jQuery и Bootstrap, существует достаточно высокая вероятность того, что у браузера клиента уже имеется кэшированная версия этих файлов.

Обращение к файлам из CDN может создать проблемы, потому что вы должны предоставить возможность резервирования, то есть возврата к версии файла, хранящейся на ваших серверах. Резервирование необходимо, потому что приложение не должно выходить из строя только из-за временной недоступности CDN. И хотя CDN обычно чрезвычайно надежны, на них тоже возможны сбои, как и у любых других сетевых сервисов, а некоторые ретивые сетевые администраторы блокируют доступ к CDN с использованием корпоративных брандмауэров.

Выполнять настройку резервирования необходимо, но это может оказаться довольно неприятным занятием. Ниже приведен пример обеспечения резервирования для популярных ресурсов Bootstrap JavaScript и CSS:

```
<link rel="stylesheet" href="//ajax.aspnetcdn.com/ajax/bootstrap/3.0.0/css/
bootstrap.min.css" />
<meta name="x-stylesheet-fallback-test" class="hidden" />
<script>!function(a,b,c){var d,e=document,f=e.getElementsByTagName("SCRIPT"),
g=f[f.length-1].previousElementSibling,h=e.defaultView&amp;&amp;e.defaultView.
getComputedStyle?e.defaultView.getComputedStyle(g):g.currentStyle;if(h&amp;&amp;
h[a]!==b)for(d=0;d<c.length;d++)e.write('<link rel="stylesheet"
href="'+c[d]+'"/>')}("visibility","hidden",["~/lib/bootstrap/css/bootstrap.
min.css"]);</script>

<script src="//ajax.aspnetcdn.com/ajax/bootstrap/3.0.0/bootstrap.min.js">
</script>
<script>(typeof($.fn.modal) === 'undefined' || document.write("<script src=\"~/lib/
bootstrap/js/bootstrap.min.js\"></script>"));</script>
```

К счастью, тег-хелперы `script` и `link` значительно упрощают определение местонахождения резервных файлов и тестов.

```
<link rel="stylesheet" href="//ajax.aspnetcdn.com/ajax/bootstrap/3.0.0/css/
bootstrap.min.css"
      asp-fallback-href="~/lib/bootstrap/css/bootstrap.min.css"
      asp-fallback-test-class="hidden"
      asp-fallback-test-property="visibility"
      asp-fallback-test-value="hidden" />

<script src="//ajax.aspnetcdn.com/ajax/bootstrap/3.0.0/bootstrap.min.js"
      asp-fallback-src="~/lib/bootstrap/js/bootstrap.min.js"
      asp-fallback-test="window.jQuery">
</script>
```

Тег-хелпер `cache`

Тег-хелпер `cache` используется для кэширования произвольных фрагментов HTML в памяти. Он предоставляет ряд параметров для управления продолжительностью хранения и для модификации ключей кэша в зависимости от контекста запроса.

Например, следующий экземпляр тег-хелпера `cache` кэширует содержимое в течение 5 минут. Для каждого уникального пользователя в системе кэшируется свой экземпляр.

```
<cache expires-sliding="@TimeSpan.FromMinutes(5)" vary-by-user="true">
    <!--Любая разметка HTML или Razor-->
    *last updated @DateTime.Now.ToLongTimeString()
</cache>
```

В более сложных ситуациях можно воспользоваться тег-хелпером `distributed-cache` для хранения фрагментов HTML в распределенном кэше. За более подробной информацией о тег-хелпере `cache` и `distributed-cache` обращайтесь по адресам <http://bit.ly/CacheTagHelper> и <http://bit.ly/distributedcachetaghelper>.

Создание тег-хелперов

По мере добавления все новых функций в приложение в файлах `cshtml` начинают проявляться закономерности. Небольшие повторяющиеся фрагменты Razor/HTML станут хорошими кандидатами для создания нестандартных тег-хелперов.

Тег-хелпер представляет собой класс, который наследует от базового класса `TagHelper` и реализует либо метод `Process`, либо `ProcessAsync`.

```
public virtual void Process(TagHelperContext context, TagHelperOutput output)
public virtual Task ProcessAsync(TagHelperContext context, TagHelperOutput output)
```

В методе `Process` тег-хелпер может проанализировать текущий объект `TagHelperContext` и сгенерировать разметку HTML или же изменить `TagHelperOutput` некоторым образом.

Очень простым примером служит кнопка на странице входа приложения Alpine Ski House для регистрации через внешнего провайдера. Хотя данная возможность вряд ли будет часто использоваться в приложении, она показывает, как можно упростить довольно сложный код Razor. Тег-хелперы эффективно работают для исключения дублирующегося кода, но они также прекрасно подходят для абстрагирования особенно сложного кода HTML и Razor. Кнопка, описанная в следующем фрагменте кода, связывается с конкретным провайдером внешнего входа.

```
<button type="submit" class="btn btn-default" name="provider" value="@provider.
AuthenticationScheme" title="Log in using your @provider.DisplayName account">
    @provider.AuthenticationScheme</button>
```

Выделим эту разметку в класс `TagHelper` с именем `LoginProviderButtonTagHelper`. Целевым элементом является `button`, тег-хелпер имеет один атрибут с именем `ski-login-provider`. По соглашению атрибуты тег-хелперов, у которых целевым является существующий элемент HTML, должны снабжаться префиксом. Это помогает отличить атрибуты тег-хелпера от обычных атрибутов HTML. Встроенные тег-хелперы ASP.NET Core используют префикс `asp-`. В приложении Alpine Ski House будет использоваться префикс `ski-`.

```
[HtmlTargetElement("button", Attributes = "ski-login-provider")]
public class LoginProviderButtonTagHelper : TagHelper
{
    [HtmlAttributeName("ski-login-provider")]
    public AuthenticationDescription LoginProvider { get; set; }

    public override void Process(TagHelperContext context, TagHelperOutput output)
    {
        output.Attributes.SetAttribute("type", "submit");
        output.Attributes.SetAttribute("name", "provider");
        output.Attributes.SetAttribute("value", LoginProvider.
            AuthenticationScheme);
        output.Attributes.SetAttribute("title", $"Log in using your
account");
        output.Attributes.SetAttribute("class", "btn btn-default");
        output.Content.SetContent(LoginProvider.AuthenticationScheme);
    }
}
```

Прежде чем вы сможете пользоваться новым тег-хелпером, необходимо сделать тег-хелперы этого проекта доступными для представлений. Для этого можно воспользоваться директивой `addTagHelper`. Чтобы применить директиву `addTagHelper` глобально, добавьте ее в файл `Views/_ViewImports.cshtml`.

```
@addTagHelper *, AlpineSkiHouse.Web
```

Теперь для упрощения описания кнопок на странице входа достаточно воспользоваться новым тег-хелпером.

```
<button ski-login-provider="provider"></button>
```

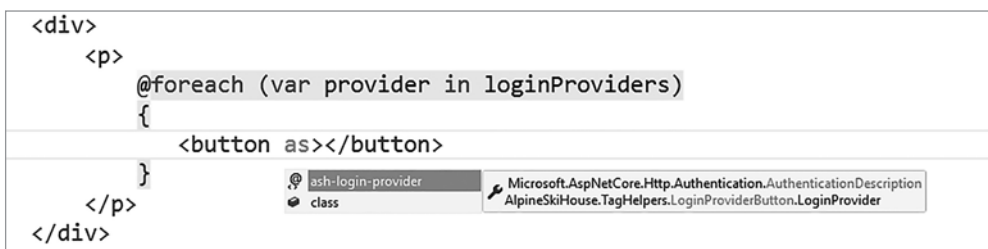


Рис. 19.2. Visual Studio с подсказкой IntelliSense для нестандартного тег-хелпера

Visual Studio предоставляет отличную поддержку IntelliSense для нестандартных тег-хелперов, как показано на рис. 19.2.

Работа с существующими атрибутами и содержимым

При создании тег-хелпера важно учитывать контекст, в котором он будет использоваться. У большинства тег-хелперов целевыми являются существующие элементы HTML, которые могут обладать атрибутами HTML. Также следует учитывать, какие атрибуты были уже добавлены в элемент. Что делать с такими атрибутами — перезаписать их или оставить в покое? Еще одна возможность — слияние текущих значений с вашими. Слияние атрибутов чаще всего используется с атрибутом `class`. В нашем примере с внешним провайдером входа тег-хелпер должен сохранить все классы, уже добавленные для кнопки, и ограничиться присоединением значения `"btn btn-default"` к значению, предоставленному пользователем. Синтаксис слияния атрибутов не отличается компактностью:

```
string classValue = "btn btn-default";
if (output.Attributes.ContainsName("class"))
{
    classValue = $"{output.Attributes["class"].Value} {classValue}";
}
output.Attributes.SetAttribute("class", classValue);
```

Код не из тех, которые бы хотелось писать снова и снова, поэтому мы выделим его в метод расширения:

```
public static class TagHelperAttributeListExtensions
{
    public static void MergeClassAttributeValue(this TagHelperAttributeList
attributes, string
newClassValue)
    {
        string classValue = newClassValue;
        if (attributes.ContainsName("class"))
        {
            classValue = $"{attributes["class"].Value} {classValue}";
        }
        attributes.SetAttribute("class", classValue);
    }
}
```

Использование метода расширения `MergeClassAttributeValue` значительно упрощает код тег-хелпера.

```
output.Attributes.MergeClassAttributeValue("btn btn-default");
```

Кроме атрибутов, также необходимо учитывать существующий HTML-контент тег-хелпера. По умолчанию Razor обрабатывает содержимое тег-хелпера, и полу-

ченная разметка HTML включается в его вывод. Тем не менее вывод тег-хелпера предоставляет возможность изменить или переписать дочерний контент при помощи свойства `Content`. Кроме `Content`, вывод тег-хелпера также предоставляет свойства `PreElement`, `PreContent`, `PostContent` и `PostElement`. Единственным фрагментом разметки HTML, который не может быть модифицирован напрямую из тег-хелпера, является родительский контент.

```
public async override Task ProcessAsync(TagHelperContext context,
                                         TagHelperOutput output)
{
    output.PreElement.SetHtmlContent("<div>Pre-Element</div>");
    output.PreContent.SetHtmlContent("<div>Pre-Content</div>");
    output.Content = await output.GetChildContentAsync();
    output.Content.Append("Adding some text after the existing content");
    output.PostContent.SetHtmlContent("<div>Post-Content</div>");
    output.PostElement.SetHtmlContent("<div>Post-Element</div>");
}
```

Вызов `GetChildContentAsync()` необходим только в том случае, если тег-хелпер вносит изменения в `Content`. Если в дочерний контент изменения не вносятся, то значение `Content` автоматически задается фреймворком.

ПРИМЕЧАНИЕ

Механизм тег-хелперов открывает эффективные возможности расширения механизма представлений Razor и предоставляет разработчикам ASP.NET новый способ генерации HTML. По адресу <https://github.com/dpaquette/taghelpersamples> можно найти ряд реальных примеров, включая более сложные ситуации с взаимодействием между родительскими и дочерними тег-хелперами.

Компоненты представлений

Компоненты представлений (view component) — еще одна конструкция MVC, позволяющая создавать виджеты для многоразового использования. В данном случае виджеты состоят из разметки Razor и служебной логики. Компоненты представлений состоят из двух частей: класса компонента представления и представления Razor.

Чтобы реализовать класс компонента представления, используйте наследование от базового класса `ViewComponent` и реализуйте метод `Invoke` или `InvokeAsync`. Этот класс может находиться в любом месте проекта, но чаще всего он размещается в папке с именем `ViewComponents`.

```
Public class MyWidgetViewComponent : ViewComponent
{
    public IviewComponentResult Invoke()
    {
```

```

        return View();
    }
}

```

Компонент представления может содержать несколько методов `Invoke`, в этом случае каждый метод содержит уникальный набор аргументов.

```

public class MyWidgetViewComponent : ViewComponent
{
    public IViewComponentResult Invoke()
    {
        return View();
    }
    public IViewComponentResult Invoke(int id)
    {
        return View();
    }
}

```

В компонентах представлений широко применяется принцип «соглашения прежде конфигурации». Если не указано обратное, имя компонента представления — это имя класса с удаленной частью `ViewComponent`. Имя `ViewComponent` может определяться явно с добавлением в класс атрибута `ViewComponent`.

```

[ViewComponent(Name = "Widget1")]
public class MyWidgetViewComponent : ViewComponent
{
    public IViewComponentResult Invoke()
    {
        return View();
    }
}

```

По аналогии с действием контроллера метод `Invoke` компонента представления возвращает представление. На этой стадии механизм представлений ищет подходящий файл `cshtml`. Если имя представления не указано явно, то по умолчанию используется файл `Views\Shared\Components\имя_компонента\Default.cshtml`, что в данном случае соответствует `Views\Shared\Components\MyWidget\Default.cshtml`.

Представление, являющееся частью компонента представления, может быть связано с моделью (по аналогии с представлениями Razor для действий контроллера).

```

public class MyWidgetViewComponent : ViewComponent
{
    public IViewComponentResult Invoke()
    {
        WidgetModel model = new WidgetModel
        {

```

```
        Name = "My Widget",  
        NumberOfItems = 2  
    }  
    return View(model);  
}  
}
```

Работа с компонентами представлений

Компоненты представлений могут быть включены в любое представление Razor; для этого следует вызвать `Component.Invoke` и указать имя компонента.

```
@await Component.InvokeAsync("MyWidget")
```

Механизм компонентов представлений находит компонент представления с подходящим именем и связывает вызов с одним из методов `Invoke` компонента представления. Для связывания с конкретной перегруженной версией метода `Invoke` следует передать аргументы в виде анонимного класса.

```
@await Component.InvokeAsync("MyWidget", new {id = 5})
```

Что случилось с дочерними действиями?

В предыдущих версиях MVC для создания компонентов или виджетов, состоящих из разметки Razor и служебной логики, использовались дочерние действия. Служебная логика реализовывалась в виде действия контроллера и обычно помечалась атрибутом `[ChildActionOnly]`. Дочерние действия были чрезвычайно полезны, но разработчик мог легко настроить их таким образом, что это имело непреднамеренные последствия для конвейера запросов.

Дочерние действия не существуют в ASP.NET Core MVC. Вместо этого для поддержки таких сценариев следует использовать новый механизм компонентов представлений. На концептуальном уровне компоненты представлений похожи на дочерние действия, но они несут меньше балласта и не имеют жизненных циклов и конвейеров, связанных с контроллерами.

Компонент представления для связи со службой поддержки

Рассмотрев основы компонентов представлений, мы переходим к реальному примеру. Представьте, что вы хотите реализовать в проекте Alpine Ski House интеграцию с приложением специалиста из службы поддержки клиентов. В главе 12 «Конфигурация и журналирование» мы создали интерфейс `ICsrInformationService` для получения информации о контакт-центре службы поддержки, включая номер телефона и количество активных операторов на данный момент. Хоро-

шо бы добавить в нижнюю часть сайта небольшую секцию, которая сообщает всю эту информацию пользователю. Использование компонента представления для реализации предоставляет хорошую возможность для взаимодействия с `ICsrInformationService`. Кроме того, оно обеспечивает некоторую гибкость, если вы захотите поэкспериментировать с размещением средств связи с контактным центром в разных местах сайта.

Класс компонента представления получает зависимость `ICsrInformationService` и реализует единственный метод `Invoke`. Если контакт-центр работает, то компонент представления получает информацию от `ICsrInformationService` и передает ее представлению `Default`. Если контакт-центр закрыт, то он возвращает представление `Closed`. Разбиение представления на две части гарантирует, что логика будет находиться в компоненте представления (вместо самого представления), что значительно упрощает тестирование компонента представления.

```
public class CallCenterStatusViewComponent : ViewComponent
{
    private readonly ICsrInformationService _csrInformationService;

    public CallCenterStatusViewComponent(ICsrInformationService
csrInformationService)
    {
        _csrInformationService = csrInformationService;
    }

    public IViewComponentResult Invoke()
    {
        if (_csrInformationService.CallCenterOnline)
        {
            var viewModel = new CallCenterStatusViewModel
            {
                OnlineRepresentatives = _csrInformationService.
OnlineRepresentatives,
                PhoneNumber = _csrInformationService.CallCenterPhoneNumber
            };
            return View(viewModel);
        }
        else
        {
            return View("Closed");
        }
    }
}
```

Представления — это просто файлы Razor. Представление `Default` связано с моделью `CallCenterStatusViewModel`, а представление `Closed` не связывается ни с какой моделью.

```
Views\Shared\Components\CallCenterStatus\Default.cshtml
@using AlpineSkiHouse.Models.CallCenterViewModels
@model CallCenterStatusViewModel

<div class="panel panel-success">
    <div class="panel-heading">Having trouble? We're here to help!</div>
    <div class="panel-body">
        We have @Model.OnlineRepresentatives friendly agents available.
        <br/>
        Give us a call at <i class="glyphicon glyphicon-earphone"></i>
            <a href="tel:@Model.
PhoneNumber">@Model.PhoneNumber</a>
    </div>
</div>
Views\Shared\Components\CallCenterStatus\Closed.cshtml
<div>The call center is closed</div>
```

Теперь этот компонент представления можно добавить в любой файл `cshtml` вашего приложения. Например, здесь он добавляется над футером `_Layout.cshtml`:

```
<div class="container body-content">
    @RenderBody()
    <hr />
    @await Component.InvokeAsync("CallCenterStatus")
    <footer>
        <p>&copy; @DateTime.Now.Year.ToString() - AlpineSkiHouse</p>
    </footer>
</div>
```

ПРИМЕЧАНИЕ

В больших организациях может быть полезно создавать компоненты представлений, предназначенные для использования в разных проектах. В этом случае компоненты представлений могут быть реализованы в отдельной библиотеке классов и загружаться при запуске приложения. Такое решение потребует лишь небольшой настройки конфигурации при запуске. Полное обучающее руководство доступно по адресу <http://aspnetmonsters.com/2016/07/2016-07-16-loading-view-components-from-a-class-library-in-asp-net-core/>.

Частичные представления

Частичное представление — это представление Razor, которое может визуализироваться в другом представлении Razor. В отличие от компонентов представлений, не существует класса с реализацией какой-либо логики или взаимодействия с его сервисами. Если класс компонента представления передает представлению данные модели представления, за передачу данных частичному представлению отвечает родительское представление. Частичные представления полезны тог-

да, когда данные уже загружены и остается только отобразить их. Независимо от того, задействованы данные модели представления или нет, вы можете использовать частичные представления для разбиения больших представлений на меньшие, более удобные части.

Шаблон проекта ASP.NET Core MVC по умолчанию включает частичное представление, которое может использоваться для генерации сценариев проверки. Такие сценарии обеспечивают проверку любых страниц, содержащих формы, на стороне клиента.

Views/Shared/_ValidationScriptsPartial.cshtml

```
<environment names="Development">
  <script src="~/lib/jquery-validation/dist/jquery.validate.js"></script>
  <script src="~/lib/jquery-validation-unobtrusive/jquery.validate.unobtrusive.
    js"></script>
</environment>
<environment names="Staging,Production">
  <script src="https://ajax.aspnetcdn.com/ajax/jquery.validate/1.14.0/jquery.
    validate.min.js"
    asp-fallback-src="~/lib/jquery-validation/dist/jquery.validate.min.js"
    asp-fallback-test="window.jQuery && window.jQuery.validator">
  </script>
  <script src="https://ajax.aspnetcdn.com/ajax/jquery.validation.
    unobtrusive/3.2.6/jquery.validate.unobtrusive.min.js"
    asp-fallback-src="~/lib/jquery-validation-unobtrusive/jquery.validate.
    unobtrusive.min.js"
    asp-fallback-test="window.jQuery && window.jQuery.validator && window.
    jQuery.validator.unobtrusive">
  </script>
</environment>
```

Любое представление, которому понадобятся эти сценарии, может включить их вызовом `@Html.Partial("_ValidationScriptsPartial")`.

Фрагмент Account/Register.cshtml

```
...
@section Scripts {
  @Html.Partial("_ValidationScriptsPartial")
}
```

Также существуют перегруженные версии с возможностью передачи модели представления, это означает, что частичное представление также может связываться с моделью, как и обычное представление. За дополнительной информацией об использовании частичных представлений обращайтесь к официальной документации ASP.NET Core по адресу <https://docs.asp.net/en/latest/mvc/views/partial.html>.

Итоги

«Строительные блоки», предоставляемые фреймворком ASP.NET Core MVC, позволяют легко разбивать приложения на компоненты, пригодные для повторного использования. Тег-хелперы могут использоваться для расширения HTML и определения атрибутов и элементов HTML, относящихся к предметной области вашего приложения; это приводит к созданию компактных, хорошо понятных файлов Razor. Компоненты представлений прекрасно подходят для определения компонентов пользовательского интерфейса, содержащих сложную логику или взаимодействующих с сервисами других приложений. Наконец, частичные представления предлагают простой механизм для определения фрагментов представлений Razor, пригодных для многократного использования. Как будет показано в главе 20 «Тестирование», и компоненты, и тег-хелперы проектировались с учетом удобства тестирования.

20

Тестирование

Балаш нажал кнопку завершения связи на дорогом Polysom-устройстве в конференц-зале.

«Итак... — сказал он. Балаш только что закончил общение с тестировщиками, и результат был не особенно радостным. — Они постоянно сталкиваются с ошибками, которые делают тестирование невозможным. Почему мы производим программный продукт столь низкого качества?»

Даниэль не согласилась с тем, что их продукт был некачественным. Проблема была в том, что продукт развивался слишком быстро для ручного тестирования. Но ведь для этого и нужны тестировщики, верно?

«Послушай, Балаш, — сказала Даниэль, — Мы ежедневно выдаем тестировщикам по десять версий. Даже если они заходят в тупик, они не остаются там надолго. Мы все исправляем с рекордной скоростью».

«Тогда почему я вижу, что те же ошибки снова возникают через неделю после их исправления?» — спросил Балаш.

Повисло неловкое молчание. Даниэль знала, что ей нечего ответить. Повторное появление ошибок после того, как они были исправлены, оправдать было невозможно.

На помощь пришел Марк-2. «Конечно, тестированию уделяется недостаточно внимания. Мы не используем разработку через тестирование — даже в первом приближении. Я знаю, все здесь хотят хорошо выполнять свою работу, но тот способ тестирования, который используется сейчас, замедляет нас, а сроков никто не отменял».

«Я знаю, — вздохнул Балаш. — Но тогда получается, что наши тестировщики просто впустую тратят время на все это. Нельзя ли по крайней мере создать автоматические тесты в тех местах, где возникают повторные ошибки?»

«Думаю, это отличная мысль, — согласилась Даниэль. — Давайте сделаем!»

Метод автоматизированного тестирования, при котором разработчики сами пишут тесты для своего кода, практически повсеместно считается самым правильным. Нередко приходится слышать, что в соответствии с методологией разработки через тестирование (TDD) тесты должны быть написаны до того, как будет написан код самого приложения. Мы не будем вступать в споры, а скажем, что в любом серьезном программном продукте должна быть реализована хотя бы минимальная форма автоматизированного тестирования. В этой главе рассматриваются некоторые подходы к тестированию различных частей приложений ASP.NET Core MVC.

Модульное тестирование

Первая разновидность тестирования, которая будет встроена в наш проект, — автоматизированное модульное тестирование. При модульном тестировании вы выполняете небольшие части приложения в изоляции от других частей и убеждаетесь в том, что они работают так, как ожидается. В контексте C# такими «модулями» обычно становятся методы или классы. Тесты пишутся с таким расчетом, чтобы их можно было легко и быстро выполнить при изменении исходного кода приложения и убедиться в том, что эти изменения не привели к непреднамеренному нарушению некоторого аспекта приложения.

Качественный набор модульных тестов повысит вашу уверенность в том, что изменение в исходном коде не вызовет регрессии в приложении. Хорошо написанные модульные тесты также в какой-то степени документируют предполагаемое поведение компонента.

XUnit

Существует несколько тестовых фреймворков для организации модульного тестирования кода C#, самыми популярными из которых являются NUnit, MSTest и xUnit.net. Несмотря на некоторые различия в функциональности, основная концепция остается неизменной. Тестовые методы помечаются специальными атрибутами. В этих тестах вы пишете код выполнения тестируемого кода и включаете некоторые условия для проверки ожидаемого результата. Эти условия либо выполняются, либо не выполняются, а система выполнения модульных тестов сообщает о результатах. Важное отличие xUnit от других фреймворков модульного тестирования заключается в том, что у xUnit нет атрибута уровня классов, используемого для пометки классов, содержащих тестовые методы. В документации xUnit приведено сравнение xUnit с другими тестовыми фреймворками (<https://xunit.github.io/docs/comparisons.html>).

Для приложения Alpine Ski House было решено использовать xUnit.net; прежде всего потому, что именно этот инструмент использовался группой ASP.NET для

модульного тестирования фреймворка ASP.NET Core, но еще и потому, что это отличный фреймворк и с ним приятно работать.

Для начала нужно создать новый проект. Строго говоря, модульные тесты C# не обязательно размещать в отдельной сборке, но обычно это стоит делать. Одна из причин заключается в том, что код модульных тестов не должен распространяться с приложением, потому что он нужен исключительно для модульного тестирования. Например, мы создали новый проект с именем `AlpineSkiHouse.Web.Test`, который содержит тесты для кода проекта `AlpineSkiHouse.Web`. В структуре папок проекты приложения располагаются в папке `src`, тогда как тестовые проекты располагаются в папке `test`. И снова соблюдать эти соглашения не обязательно, но они соблюдаются во многих приложениях .NET.

Кроме того, в тестовый проект необходимо включить ссылки на два пакета: `xunit` и `dotnet-test-xunit`. Свойству `testRunner` также должно быть присвоено значение `xunit`; и конечно, не забудьте добавить ссылку на тестируемый проект, в данном случае `AlpineSkiHouse.Web`.

```
"dependencies": {
  "AlpineSkiHouse.Web": "1.0.0-*",
  "dotnet-test-xunit": "1.0.0-rc2-build10025",
  "xunit": "2.1.0"
},
"testRunner": "xunit"
```

Основы xUnit

В xUnit модульный тест обозначается атрибутом метода. Допускается использование атрибутов двух типов: `Fact` и `Theory`. Первый (факт) обозначает условие, которое всегда истинно, а второй (теория) — тест, который должен быть истинным для некоторого набора входных данных. Атрибут `Theory` может использоваться для тестирования нескольких условий одним тестовым методом.

```
public class SimpleTest
{
  [Fact]
  public void SimpleFact()
  {
    var result = Math.Pow(2, 2);
    Assert.Equal(4, result);
  }

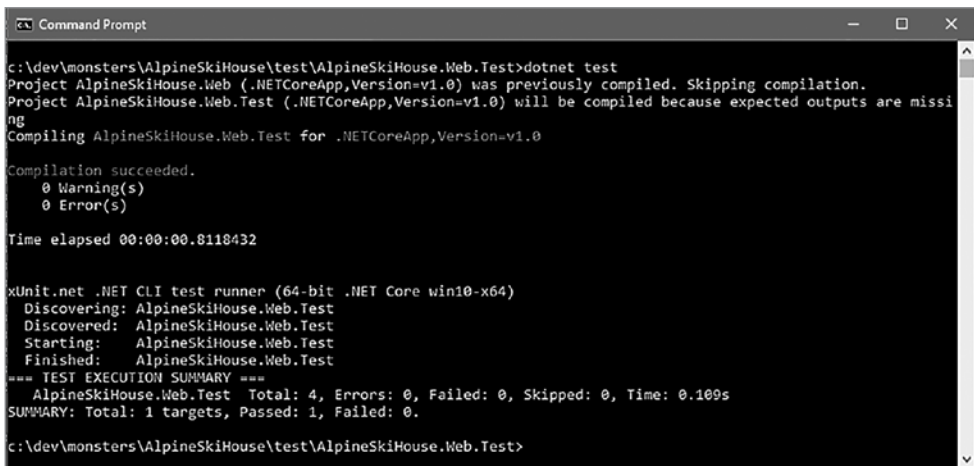
  [Theory]
  [InlineData(2, 4)]
  [InlineData(4, 16)]
  [InlineData(8, 64)]
  public void SimpleTheory(int amount, int expected)
  {
```

```
        var result = Math.Pow(amount, 2);
        Assert.Equal(expected, result);
    }
}
```

За дополнительной информацией обращайтесь к официальной документации [xunit.net](https://xunit.github.io) по адресу <https://xunit.github.io>.

Запуск тестов

Для запуска модульных тестов проще всего воспользоваться инструментом командной строки `dotnet`. В папке, содержащей тестовый проект, выполните команду `dotnet test`; команда компилирует тестовый проект и запускает все тесты, как показано на рис. 20.1.



```
Command Prompt
c:\dev\monsters\AlpineSkiHouse\test\AlpineSkiHouse.Web.Test>dotnet test
Project AlpineSkiHouse.Web (.NETCoreApp,Version=v1.0) was previously compiled. Skipping compilation.
Project AlpineSkiHouse.Web.Test (.NETCoreApp,Version=v1.0) will be compiled because expected outputs are missing
Compiling AlpineSkiHouse.Web.Test for .NETCoreApp,Version=v1.0
Compilation succeeded.
    0 Warning(s)
    0 Error(s)
Time elapsed 00:00:00.8118432

xUnit.net .NET CLI test runner (64-bit .NET Core win10-x64)
  Discovering: AlpineSkiHouse.Web.Test
  Discovered:  AlpineSkiHouse.Web.Test
  Starting:    AlpineSkiHouse.Web.Test
  Finished:    AlpineSkiHouse.Web.Test
=== TEST EXECUTION SUMMARY ===
   AlpineSkiHouse.Web.Test  Total: 4, Errors: 0, Failed: 0, Skipped: 0, Time: 0.109s
SUMMARY: Total: 1 targets, Passed: 1, Failed: 0.
c:\dev\monsters\AlpineSkiHouse\test\AlpineSkiHouse.Web.Test>
```

Рис. 20.1. Запуск модульных тестов из командной строки командой `dotnet test`

Средства командной строки также упрощают интеграцию с существующими сценариями сборки. Все, что от вас потребуется, — добавить шаг с выполнением тестов.

Другой вариант — запуск модульных тестов прямо из Visual Studio с использованием встроенного окна Test Explorer. Это интегрированное решение для разработчиков предоставляет возможность запуска тестов с присоединением отладчика.

Чтобы открыть окно Test Explorer в Visual Studio, выберите команду **Test ► Windows ► Test Explorer** в главном меню. В окне Test Explorer (рис. 20.2) должен выводиться список всех тестов в вашем решении. Если тесты отсутствуют в списке,

возможно, вам стоит пересобрать решение, пока окно Test Explorer остается открытым.

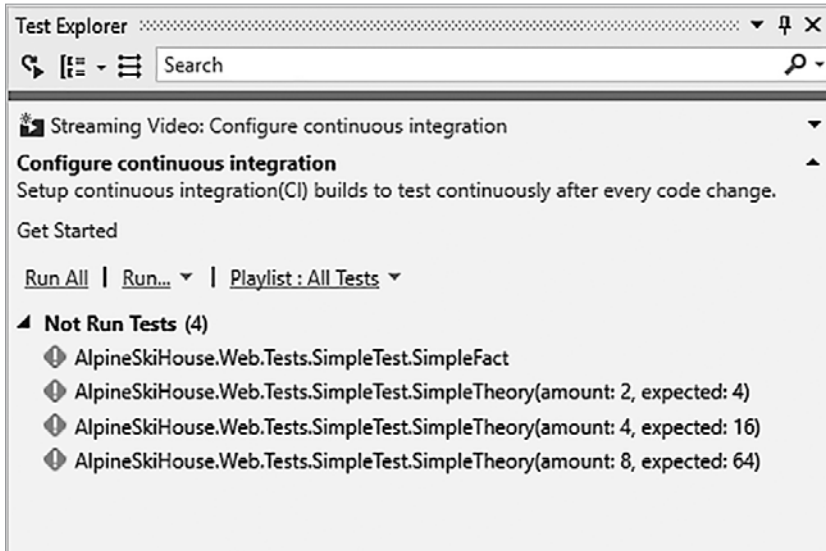


Рис. 20.2. Окно Test Explorer в Visual Studio

Щелкните на ссылке **Run All**, чтобы запустить все тесты. Успешно прошедшие тесты будут помечены зеленой галочкой, а неуспешно — красным крестиком. Если щелкнуть на неуспешно прошедшем тесте, вы получите сводку с информацией, включая ссылку на метод, в котором произошел сбой, и конкретно невыполненное условие.

НЕПРЕРЫВНОЕ ТЕСТИРОВАНИЕ

В окне Test Explorer предусмотрена малоизвестная функция для организации непрерывного тестирования. В левой верхней части панели инструментов Test Explorer расположен переключатель **Run Tests After Build**. Когда он находится во включенном состоянии, Visual Studio автоматически перезапускает тесты после сборки проекта. Эта функция позволяет почти немедленно получить обратную связь, если в результате внесения изменений будут нарушены какие-либо из существующих тестов.

Программистам, использующим другой редактор вместо Visual Studio, может пригодиться задача **dotnet watch** по адресу <https://github.com/aspnet/DotNetTools/tree/dev/src/Microsoft.DotNet.Watcher.Tools>. Задача **watch** перекомпилирует и перезапускает тесты каждый раз, когда она обнаруживает изменение в файлах исходного кода в проекте.

Организация модульного тестирования

Мы обсудили преимущества автоматического модульного тестирования, но при этом необходимо учитывать затраты, связанные с сопровождением модульных тестов. Ведь модульные тесты сами по себе содержат программный код и требуют такого же обращения, как и код рабочей версии приложения. Без стандартных соглашений об именах и стиля программирования сопровождение модульных тестов может обернуться сущим кошмаром.

При выборе имен и в организации тестирования мы стараемся соблюдать некоторые правила. Во-первых, структура папок тестового проекта должна соответствовать структуре папок тестируемой сборки. Организация файлов проекта более подробно рассматривается в главе 24 «Организация кода».

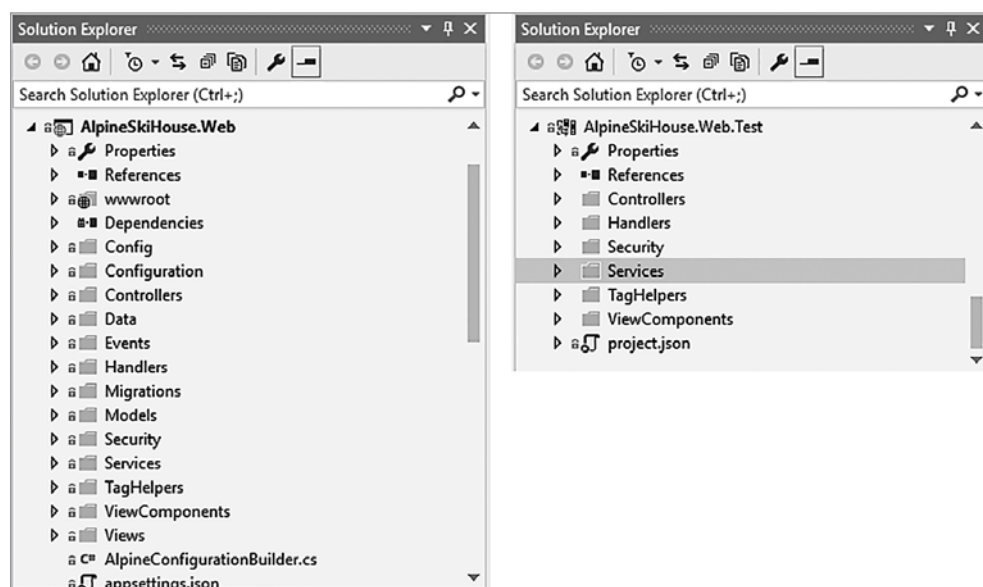


Рис. 20.3. Сравнение структуры папок в проектах AlpineSkiHouse.Web и AlpineSkiHouse.Web.Test

Все тесты, относящиеся к некоторому классу, собираются в одном файле. Иначе говоря, тесты `CsrInformationService` будут находиться в файле с именем `CsrInformationServiceTests.cs` (см. листинг 20.1). Внутри тестового класса тесты упорядочиваются при помощи вложенных классов. Каждый внутренний класс специализируется на тестировании в определенном сценарии. Например, в случае `CsrInformationService` необходимо протестировать сценарий с отсутствием операторов в контакт-центре. Этот сценарий проверяется внутренним классом `GivenThereIsAtLeastOneRepresentativeOnline` в листинге 20.1. Класс

`CsrInformationService` имеет зависимость `IOptions<CsrInformationOptions>`, которая инициализируется в конструкторе внутреннего класса. Наконец, тестовый метод проверяет ожидания текущей ситуации. В простейшем случае проверяется, что свойство `CallCenterOnline` равно `false`. Содержимое файла `CsrInformationServiceTests.cs` приведено в листинге 20.1.

Листинг 20.1. Модульные тесты для `CsrInformationService`

```
public class CsrInformationServiceTests
{
    public class GivenThereAreNoRepresentativesOnline
    {
        private IOptions<CsrInformationOptions> options;

        public GivenThereAreNoRepresentativesOnline()
        {
            options = Options.Create(new CsrInformationOptions());
            options.Value.OnlineRepresentatives = 0;
        }

        [Fact]
        public void CallCenterOnlineShouldBeFalse()
        {
            var service = new CsrInformationService(options);
            Assert.False(service.CallCenterOnline);
        }
    }

    public class GivenThereIsAtLeastOneRepresentativeOnline
    {
        public static readonly List<object[]> options = new List<object[]>
        {
            new object[]
            {
                Options.Create(new CsrInformationOptions { OnlineRepresentatives =
                                                                    1 } )},
            new object[]
            {
                Options.Create(new CsrInformationOptions { OnlineRepresentatives =
                                                                    2 } )},
            new object[]
            {
                Options.Create(new CsrInformationOptions { OnlineRepresentatives =
                                                                    3 } )},
            new object[]
            {
                Options.Create(new CsrInformationOptions { OnlineRepresentatives =
                                                                    1000 }
            }},
            new object[]
            {
                Options.Create(new CsrInformationOptions { OnlineRepresentatives =
                                                                    100000 }
            }
        }
    }
}
```

```

    };

    [Theory]
    [InlineData(nameof(options))]
    public void CallCenterOnlineShouldBeTrue(IOptions<CsrInformationOptions>
                                           options)
    {
        var service = new CsrInformationService(options);
        Assert.True(service.CallCenterOnline);
    }

    [Theory]
    [InlineData(nameof(options))]
    public void
OnlineRepresentativesShouldMatchOptionsSource(IOptions<CsrInformationOptions>
                                              options)
    {
        var service = new CsrInformationService(options);
        Assert.Equal(options.Value.OnlineRepresentatives,
                     service.OnlineRepresentatives);
    }
}

```

В листинге 20.1 также тестируется второй сценарий во внутреннем классе `GivenThereIsAtLeastOneRepresentativeOnline`. На этот раз тест проверяет `CsrInformationService` при наличии хотя бы одного оператора. Обратите внимание на использование атрибутов `Theory` и `InlineData` для тестирования с набором возможных входных значений, соответствующих количеству операторов. Свойство `options` предоставляет массив возможных значений `CsrInformationOptions` с разными числовыми значениями `OnlineRepresentatives`. Этот сценарий включает два теста, помеченных атрибутом `Theory`. Первый проверяет, что свойство `CallCenterOnline` возвращает `true`, а второй проверяет, что свойство `OnlineRepresentatives` равно ожидаемому значению.

Тестирование контроллеров

В идеале контроллеры должны содержать минимум логики, вследствие чего они не нуждаются в особом тестировании. Контроллер должен отвечать только за преобразование между веб-концепциями (такими, как запросы и ответы HTTP) и концепциями приложения. Большая часть реальной работы должна быть реализована в сервисах, командах и обработчиках событий и других местах кода, более подходящих для многократного использования. С другой стороны, тестирование контроллеров для проверки того, что сервисы используются так, как ожидается, и возвращаются правильные ответы HTTP, тоже имеет смысл. Например, полезно протестировать контроллер для проверки того, что неавторизованный пользователь не сможет получить доступ к ограниченному ресурсу.

Рассмотрим контроллер `SkiCardController` из главы 10 «Entity Framework Core». Контроллер `SkiCardController` имеет зависимости `SkiCardContext`, `UserManager<ApplicationUser>` и `IAuthorizationService`. Чем больше зависимостей, тем сложнее подготовка. В некоторых случаях желательно использовать прием макетирования (mocking) с передачей фиктивной версии зависимости проверки того, как контроллер взаимодействует только с частью этого интерфейса. Мы будем использовать макетирование для тестирования взаимодействия `SkiCardController` с `IAuthorizationService`. Для некоторых зависимостей, таких как `SkiCardContext`, макетирование может быть затруднено. В случае `DbContext` из Entity Framework (например, `SkiCardContext`) намного проще создать конкретный экземпляр контекста на основе хранилища данных в памяти вместо полноценной базы данных. Из этого следует вынести один важный урок: не все задачи решаются при помощи макетирования. Будьте практичны при тестировании и делайте то, что лучше подходит для конкретной ситуации.

Чтобы создать экземпляр `SkiCardContext` на основе базы данных в памяти, необходимо передать параметры, отличные от тех, что используются приложением во время выполнения. Созданием таких параметров обычно занимается фреймворк внедрения зависимостей, но в проекте модульного тестирования этим придется заниматься самостоятельно.

Чтобы избежать повторения кода, мы создали фабричный метод. Он может использоваться для любого экземпляра `DbContext`:

```
public static class InMemoryDbContextOptionsFactory
{
    public static DbContextOptions<T> Create<T>() where T : DbContext
    {
        var serviceProvider = new ServiceCollection()
            .AddEntityFrameworkInMemoryDatabase()
            .BuildServiceProvider();

        var builder = new DbContextOptionsBuilder<T>();
        builder.UseInMemoryDatabase()
            .UseInternalServiceProvider(serviceProvider);
        return builder.Options;
    }
}
```

Начнем с тестирования сценария, при котором действие `HttpPost Edit` вызывается для объекта `SkiCard`, не существующего в контексте. В этом случае метод действия должен возвращать `NotFoundResult`. Зависимости `UserManager` и `IAuthorizationService` в этом сценарии даже не используются, поэтому в этом конкретном тесте можно просто передать для них значения `null`.

```
public class WhenEditingASkiCardThatDoesNotExistInTheContext
{
    [Fact]
```

```

public async Task EditActionShouldReturnNotFound()
{
    using (SkiCardContext context =
        new SkiCardContext(InMemoryDbContextOptionsFactory.
            Create<SkiCardContext>()))
    {
        var controller = new SkiCardController(context, null, null);
        var result = await controller.Edit(new EditSkiCardViewModel
        {
            Id = 2,
            CardHolderFirstName = "Dave",
            CardHolderLastName = "Paquette",
            CardHolderBirthDate = DateTime.Now.AddYears(-99),
            CardHolderPhoneNumber = "555-123-1234"
        });
        Assert.IsType<NotFoundResult>(result);
    }
}

```

В другом, чуть более сложном сценарии злоумышленник пытается вызвать метод `HttpPost Edit` для редактирования экземпляра `SkiCard`, не принадлежащего ему. Этот сценарий тестируется в классе `GivenAHackerTriesToEditSomeoneElsesSkiCard` (листинг 20.2). Он имеет две зависимости, а также требует подготовки информации в `ControllerContext`, а именно объекта `User`, извлекаемого из `HttpContext`. Этот подготовительный код перемещается в конструктор, чтобы тестовый метод оставался простым и удобочитаемым. Сначала создается `SkiCardContext` и добавляется экземпляр `SkiCard`, который пытается редактировать злоумышленник. Затем создается `ControllerContext` с `DefaultHttpContext` и новым экземпляром `ClaimsPrincipal`. Задавать какие-либо значения в `ClaimsPrincipal` не обязательно, потому что вы хотите убедиться только в том, что `ClaimsPrincipal` правильно передается `IAuthorizationService`. В тестовом методе создается контроллер и настраивается ожидаемое обращение к `IAuthorizationService`. Конкретно ожидается вызов метода `AuthorizeAsync` с передачей `_badGuyPrincipal` и `_skiCard`, с проверкой `EditSkiCardAuthorizationRequirement`. Для подготовки и проверки ожидаемых взаимодействий между `SkiCardController` и `IAuthorizationService` мы используем `Moq` — простую, но мощную библиотеку макетирования для .NET. За дополнительной информацией о `Moq` обращайтесь по адресу <https://github.com/Moq/moq4>.

Листинг 20.2. Пример тестирования метода действия `Edit` контроллера `SkiCardController`

```

public class GivenAHackerTriesToEditSomeoneElsesSkiCard : IDisposable
{
    SkiCardContext _skiCardContext;
    SkiCard _skiCard;
    ControllerContext _controllerContext;
}

```

```

ClaimsPrincipal _badGuyPrincipal;
Mock<IAuthorizationService> _mockAuthorizationService;

public GivenAHackerTriesToEditSomeoneElsesSkiCard()
{
    _skiCardContext =
        new SkiCardContext(InMemoryDbContextOptionsFactory.
                           Create<SkiCardContext>());
    _skiCard = new SkiCard
    {
        ApplicationUserId = Guid.NewGuid().ToString(),
        Id = 5,
        CardHolderFirstName = "James",
        CardHolderLastName = "Chambers",
        CardHolderBirthDate = DateTime.Now.AddYears(-150),
        CardHolderPhoneNumber = "555-555-5555",
        CreatedOn = DateTime.UtcNow
    };

    _skiCardContext.SkiCards.Add(_skiCard);
    _skiCardContext.SaveChanges();

    _badGuyPrincipal = new ClaimsPrincipal();
    _controllerContext = new ControllerContext()
    {
        HttpContext = new DefaultHttpContext
        {
            User = _badGuyPrincipal
        }
    };

    _mockAuthorizationService = new Mock<IAuthorizationService>();
}

[Fact]
public async void EditActionShouldReturnChallengeResult()
{
    var controller = new SkiCardController(_skiCardContext, null,
    _mockAuthorizationService.Object)
    {
        ControllerContext = _controllerContext
    };

    _mockAuthorizationService.Setup(
        a => a.AuthorizeAsync(
            _badGuyPrincipal,
            _skiCard,
            It.IsAny<IEnumerable<IAuthorizationRequirement>>>(
                r => r.Count() == 1 && r.First() is
EditSkiCardAuthorizationRequirement)))

```

```

        .Returns(Task.FromResult(false));

var result = await controller.Edit(new EditSkiCardViewModel
{
    Id = _skiCard.Id,
    CardHolderFirstName = "BadGuy",
    CardHolderLastName = "McHacker",
    CardHolderBirthDate = DateTime.Now.AddYears(-25),
    CardHolderPhoneNumber = "555-555-5555"
});

Assert.IsType<ChallengeResult>(result);
_mockAuthorizationService.VerifyAll();
}
public void Dispose()
{
    _skiCardContext.Dispose();
}
}

```

xUnit создает новый экземпляр тестового класса для каждого выполняемого теста. Если класс имеет пометку `IDisposable`, xUnit также вызывает метод `Dispose` после каждого теста. В сценарии из листинга 20.2 класс имеет пометку `IDisposable`, и в метод `Dispose` добавляется логика завершения, которая обеспечивает уничтожение экземпляра `_skiCardContext` после выполнения теста. Есть много других условий, которые можно (и нужно) было бы проверить в листинге 20.2. Например, в приведенном выше сценарии следовало бы включить тест для проверки того, что карта не была модифицирована. Еще лучше следовало бы переместить большую часть взаимодействия с `SkiCardContext` в команду, чтобы код контроллера проще тестировался. За дополнительной информацией о рефакторинге и паттерне «Команда» обращайтесь к главе 23 «Рефакторинг и повышение качества кода».

Тег-хелперы для тестирования

Тег-хелперы для модульного тестирования предоставляют возможность протестировать разметку HTML, выводимую тег-хелпером в разных сценариях. Если не считать некоторых хлопот с подготовкой, тег-хелперы для тестирования достаточно просты. По сути вы должны подготовить `TagHelperContext`, вызвать `Process` или `ProcessAsync` для тег-хелпера и проверить, что в `TagHelperOutput` были внесены ожидаемые изменения. Основные сложности связаны с созданием `TagHelperContext` и `TagHelperOutput` для тестирования.

Так как ASP.NET Core MVC распространяется с открытым кодом, если вы окажетесь в тупике, можно просто изучить исходный код. В данном случае мы проверили, как группа ASP.NET тестировала встроенные тег-хелперы. Через несколько минут возни с исходными кодами на GitHub мы нашли решение с написанием пары статических методов для создания `TagHelperContext` и `TagHelperOutput`.


```

private static TagHelperContext GetTagHelperContext(string id = "testid")
{
    return new TagHelperContext(
        allAttributes: new TagHelperAttributeList(),
        items: new Dictionary<object, object>(),
        uniqueId: id);
}

private static TagHelperOutput GetTagHelperOutput(
    string tagName = "button",
    TagHelperAttributeList attributes = null,
    string childContent = "")
{
    attributes = attributes ?? new TagHelperAttributeList();
    return new TagHelperOutput(
        tagName,
        attributes,
        getChildContentAsync: (useCachedResult, encoder) =>
        {
            var tagHelperContent = new DefaultTagHelperContent();
            tagHelperContent.SetHtmlContent(childContent);
            return Task.FromResult<TagHelperContent>(tagHelperContent);
        });
}

```

Эти методы специфичны для каждого тестируемого тег-хелпера. Например, при тестировании некоторых тег-хелперов необходимо добавить атрибуты по умолчанию к `TagHelperContext` и `TagHelperOutput`. В данном случае мы добавили эти статические методы в класс `LoginProviderButtonTagHelperTests`, чтобы они могли использоваться для всех сценариев с тег-хелперами. В листинге 20.3 приведен один из сценариев из класса `LoginProviderButtonTagHelperTests`.

Листинг 20.3. Модульные тесты для `LoginProviderButtonTagHelper`

```

public class WhenTargettingAnEmptyButtonTag
{
    TagHelperContext _context;
    TagHelperOutput _output;
    AuthenticationDescription _loginProvider;
    LoginProviderButtonTagHelper _tagHelper;

    public WhenTargettingAnEmptyButtonTag()
    {
        _loginProvider = new AuthenticationDescription
        {
            DisplayName = "This is the display name",
            AuthenticationScheme = "This is the scheme"
        };

        _tagHelper = new LoginProviderButtonTagHelper()
    }
}

```

```

        {
            LoginProvider = _loginProvider
        };

        _context = GetTagHelperContext();
        _output = GetTagHelperOutput();
    }

    [Fact]
    public void TheTypeAttributeShouldBeSetToSubmit()
    {
        _tagHelper.Process(_context, _output);

        Assert.True(_output.Attributes.ContainsName("type"));
        Assert.Equal("submit", _output.Attributes["type"].Value);
    }

    [Fact]
    public void TheNameAttributeShouldBeSetToProvider()
    {
        _tagHelper.Process(_context, _output);

        Assert.True(_output.Attributes.ContainsName("name"));
        Assert.Equal("provider", _output.Attributes["name"].Value);
    }

    [Fact]
    public void TheValueAttributeShouldBeTheAuthenticationScheme()
    {
        _tagHelper.Process(_context, _output);

        Assert.True(_output.Attributes.ContainsName("value"));
        Assert.Equal(_loginProvider.AuthenticationScheme, _output.
            Attributes["value"].
Value);
    }

    [Fact]
    public void TheTitleAttributeShouldContainTheDisplayName()
    {
        _tagHelper.Process(_context, _output);

        Assert.True(_output.Attributes.ContainsName("title"));
        Assert.Contains(_loginProvider.DisplayName, _output.Attributes["title"].
            Value.
ToString());
    }

    [Fact]
    public void TheContentsShouldBeSetToTheAuthenticationScheme()
    {

```

```

        _tagHelper.Process(_context, _output);

        Assert.Equal(_loginProvider.AuthenticationScheme, _output.Content.
            GetContent());
    }
}

```

Сценарий в листинге 20.3 тестирует ожидаемый вывод `LoginProviderButtonTagHelper` для кнопки, которая не содержит никакой внутренней разметки HTML и не имеет установленных атрибутов HTML. `TagHelperContext`, `TagHelperOutput` и `LoginProviderButtonTagHelper` инициализируются в конструкторе. Каждый отдельный тестовый метод вызывает метод `Process` тег-хелпера и проверяет, что для `TagHelperOutput` был правильно установлен соответствующий атрибут. Также необходимо протестировать `LoginProviderButtonTagHelper` в сценариях, в которых кнопка имеет существующие атрибуты HTML и внутреннюю разметку HTML. Эти дополнительные сценарии представлены в репозитории GitHub приложения Alpine Ski House: <https://github.com/AspNetMonsters/AlpineSkiHouse/>.

Тестирование компонентов представлений

Тестирование компонентов представлений имеет много общего с тестированием контроллера. Вы подготавливаете или макетируете все необходимые зависимости, вызываете метод `Invoke` и проверяете результат. В листинге 20.4 приведен пример тестирования компонента `CallCenterStatusViewComponent`.

Листинг 20.4. Модульные тесты для `CallCenterStatusViewComponent`

```

public class CallCenterStatusViewComponentTests
{
    public class GivenTheCallCenterIsClosed
    {
        [Fact]
        public void TheClosedViewShouldBeReturned()
        {
            var _csrInfoServiceMock = new Mock<ICsrInformationService>();
            _csrInfoServiceMock.Setup(c => c.CallCenterOnline).Returns(false);

            var viewComponent = new CallCenterStatusViewComponent
                (_csrInfoServiceMock.Object);

            var result = viewComponent.Invoke();

            Assert.IsType<ViewViewComponentResult>(result);
            var viewResult = result as ViewViewComponentResult;
            Assert.Equal("Closed", viewResult.ViewName);
        }
    }
}

```

```

public class GivenTheCallCenterIsOpen
{
    [Fact]
    public void TheDefaultViewShouldBeReturned()
    {
        var _csrInfoServiceMock = new Mock<ICsrInformationService>();
        _csrInfoServiceMock.Setup(c => c.CallCenterOnline).Returns(true);

        var viewComponent = new CallCenterStatusViewComponent
            (_csrInfoServiceMock.Object);

        var result = viewComponent.Invoke();

        Assert.IsType<ViewViewComponentResult>(result);
        var viewResult = result as ViewViewComponentResult;
        Assert.Null(viewResult.ViewName);
    }
}

```

В листинге 20.4 для тестирования `CallCenterStatusViewComponent` используется макет `ICsrInformationService`. Первый сценарий проверяет, что если свойство `CallCenterOnline` экземпляра `ICsrInformationService` равно `false`, то возвращается представление `Closed`. Второй сценарий проверяет, что для истинного свойства `CallCenterOnline` возвращается представление по умолчанию.

Провести модульное тестирование кода C# в приложении ASP.NET Core намного проще, чем объяснить руководству, почему ваше приложение случайно выдало бесплатные абонементы на \$50 000.

Тестирование кода JavaScript

Как было показано в главе 15 «Роль JavaScript», кроме кода C#, наше приложение содержит заметный объем кода JavaScript. Сложность кода на стороне клиента в большинстве приложений нуждается в тестировании, но во многих приложениях ASP.NET об этом забывают. К счастью, существуют отличные инструменты для тестирования JavaScript, работающие примерно по тому же принципу, что и xUnit.net.

Jasmine

Как это часто бывает при работе с JavaScript, существует как минимум десяток конкурирующих фреймворков модульного тестирования. Доскональное сравнение этих фреймворков выходит за рамки книги. Для проекта Apline Ski House мы выбрали Jasmine, потому что в прошлом чаще всего работали именно с этим инструментом. Jasmine устанавливается при помощи npm.

```
npm install --save-dev jasmine
npm install -g jasmine
```

По действующим соглашениям Jasmine помещает файлы в папку с именем `spec`. В отличие от тестов C#, тесты JavaScript лучше размещать в одном проекте с исходным кодом JavaScript (в данном случае это проект `AlpineSkiHouse.Web`). Размещение тестов JavaScript в отдельном проекте создает ненужные сложности и значительно затрудняет выполнение тестов. Так как сами тесты являются файлами JavaScript и не встраиваются в сборку, их можно легко исключить из процесса публикации. Иначе говоря, мы можем позаботиться о том, чтобы предотвратить случайное распространение кода тестов как части приложения.

Модульные тесты в Jasmine реализуются вложением функций. Каждый файл начинается с вызова глобальной функции Jasmine `describe`, которой передаются текст с описанием тестируемого сценария и функция, реализующая этот сценарий. Внутри функции реализации можно реализовать несколько тестов, вызывая функцию Jasmine `it` с передачей описания теста и реализующей функции. Функция `expect` используется для описания ожиданий по аналогии с тем, как вы используете `Assert` в `xUnit.net`.

```
describe("The scenario we are testing", function() {
    var someValue;
    it("should be true", function() {
        a = true;
        expect(a).toBe(true);
    });
});
```

Допускается вложение функций `describe`; кроме того, Jasmine предоставляет ряд полезных подготовительных и завершающих функций: `beforeEach`, `afterEach`, `beforeAll` и `afterAll`. Превосходная документация Jasmine доступна по адресу <http://jasmine.github.io/>.

Организация тестирования JavaScript

Тесты JavaScript в Jasmine по возможности организуются по тем же принципам, что и модульные тесты C#. А именно структура папок в папке `specs` отражает структуру папки `Scripts`, в которой располагается весь код JavaScript приложения. Например, тесты для файла `Controls/MetersSkied.js` находятся в файле `specs/Controls/MetersSkiedTest.js`.

Каждый тестовый файл начинается с описания высокоуровневого сценария и загрузки тестируемого модуля. В следующем примере используется модуль `Controls/MetersSkied`. Загрузка выполняется в функции Jasmine `beforeAll`. Важно выдать сигнал о завершении импортирования вызовом функции `done()`. Далее следуют тесты и вложенные сценарии.

```
describe("When viewing the MetersSkied chart", function () {
    var MetersSkied;
    beforeAll(function (done) {
        System.import("Controls/MetersSkied").then(function (m) {
            MetersSkied = m;
            done();
        });
    });
    // Вложенные сценарии или тесты
    it("should display data for the ski season", function () {
        var div = document.createElement("div");
        var chart = new MetersSkied.MetersSkied(div);

        expect(chart.labels.length).toBe(6);
        expect(chart.labels).toContain("November");
        expect(chart.labels).toContain("December");
        expect(chart.labels).toContain("January");
        expect(chart.labels).toContain("February");
        expect(chart.labels).toContain("March");
        expect(chart.labels).toContain("April");
    })
});
```

Выполнение тестов

Jasmine включает механизм запуска модульных тестов, но он в основном ориентирован на приложения Node.js, а не на приложения, выполняемые в браузере. К счастью, существуют другие механизмы запуска модульных тестов, разработанные для выполнения модульных тестов Jasmine в браузере. Для веб-приложений (таких, как AlpineSkiHouse.Web) хорошо подходит исполнитель тестов Карма, потому что он позволяет выполнять тесты в контексте разных браузеров (таких, как Chrome, FireFox, Internet Explorer и PhantomJS). Карма берет на себя инициализацию тестов Jasmine (в данном контексте под инициализацией понимается создание страницы HTML, включающей соответствующие файлы JavaScript, и запуск веб-сервера для предоставления этих страниц браузеру). Кроме того, Карма обеспечивает выполнение тестов в выбранных вами браузерах и легко устанавливается с использованием npm.

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

После того как пакет Карма будет установлен, выполните в командной строке команду `karma init`; эта команда задает серию вопросов о том, какие файлы JavaScript следует включить и для каких браузеров следует проводить тестирование. Получив ответ на все вопросы, Карма создает файл `karma.conf.js`.

```
// Конфигурация Karma
module.exports = function(config) {
  config.set({
    // Базовый путь, который будет использоваться для разрешения всех путей
    basePath: 'wwwroot',
    frameworks: ['jasmine'],

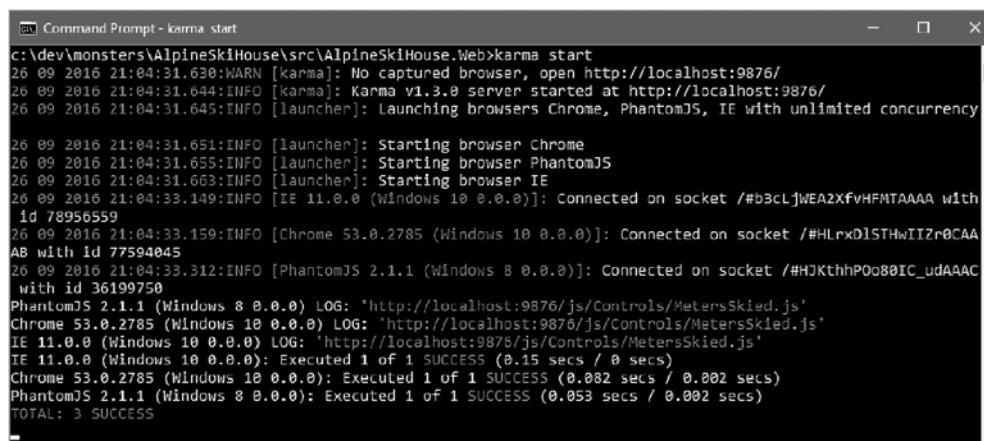
    // Список файлов/шаблонов для загрузки в браузер
    files: [
      'js/system.js',
      'js/system-polyfills.js',
      'js/jspmconfig.js',
      { pattern: 'js/**/*.js', included: false, watched: true, served: true },
      // 'js/Controls/MetersSkied.js',
      '../spec/**/*.Tests.js'
    ],
    reporters: ['progress'],
    // Порт веб-сервера
    port: 9876,
    colors: true,
    logLevel: config.LOG_INFO,
    // Разрешение/запрет отслеживания файлов и исполнения тестов
    // при обнаружении изменений
    autoWatch: true,
    browsers: ['Chrome', 'PhantomJS', 'IE'],
    singleRun: false,
    concurrency: Infinity,
    proxies: {
      '/js': '/base/js'
    }
  })
}
```

Свойство `files` играет важную роль. Сначала должны загружаться файлы `system.js` и `system-polyfills.js`, а затем `jspmconfig.js`. Они выполняют начальные действия, необходимые для загрузки модулей в наших тестах. Затем мы указываем шаблон `*.js`, обозначающий все файлы в папке `js`, но при этом сообщаем Карма, что эти файлы не должны включаться автоматически. Веб-сервер Карма должен предоставлять отдельные файлы JavaScript только тогда, когда они будут запрашиваться загрузчиком модулей `System.js`.

Для начала мы настроили Карма для выполнения тестов для Chrome, IE и PhantomJS. PhantomJS является консольным браузером, это означает, что тесты для него можно запускать без открытия окна браузера. Для каждого заданного браузера устанавливается плагин `karma-*-launcher`. Также существуют плагины для Edge, Firefox и других браузеров.

```
npm install --save-dev karma-chrome-launcher
npm install --save-dev karma-phantomjs-launcher
npm install --save-dev karma-ie-launcher
```

После завершения настройки Карма запустите модульные тесты командой `karma start` из командной строки. Карма запускает браузеры Chrome и Internet Explorer и выполняет все найденные модульные тесты Jasmine. Результаты выводятся в командной строке (рис. 20.4).



```
Command Prompt - karma start
c:\dev\monsters\AlpineSkiHouse\src\AlpineSkiHouse.Web>karma start
26 09 2016 21:04:31.630:WARN [karma]: No captured browser, open http://localhost:9876/
26 09 2016 21:04:31.644:INFO [karma]: Karma v1.3.0 server started at http://localhost:9876/
26 09 2016 21:04:31.645:INFO [launcher]: Launching browsers Chrome, PhantomJS, IE with unlimited concurrency
26 09 2016 21:04:31.651:INFO [launcher]: Starting browser Chrome
26 09 2016 21:04:31.655:INFO [launcher]: Starting browser PhantomJS
26 09 2016 21:04:31.663:INFO [launcher]: Starting browser IE
26 09 2016 21:04:33.149:INFO [IE 11.0.0 (Windows 10 0.0.0)]: Connected on socket /#b3cljWEA2XfvHFMtAAAA with
id 78950559
26 09 2016 21:04:33.159:INFO [Chrome 53.0.2785 (Windows 10 0.0.0)]: Connected on socket /#HLrxD1SThwIIZr0CAA
AB with id 77594045
26 09 2016 21:04:33.312:INFO [PhantomJS 2.1.1 (Windows 8 0.0.0)]: Connected on socket /#HJKthhP0o80IC_udAAAC
with id 36199750
PhantomJS 2.1.1 (Windows 8 0.0.0) LOG: 'http://localhost:9876/js/Controls/MetersSkied.js'
Chrome 53.0.2785 (Windows 10 0.0.0) LOG: 'http://localhost:9876/js/Controls/MetersSkied.js'
IE 11.0.0 (Windows 10 0.0.0) LOG: 'http://localhost:9876/js/Controls/MetersSkied.js'
IE 11.0.0 (Windows 10 0.0.0): Executed 1 of 1 SUCCESS (0.15 secs / 0 secs)
Chrome 53.0.2785 (Windows 10 0.0.0): Executed 1 of 1 SUCCESS (0.082 secs / 0.002 secs)
PhantomJS 2.1.1 (Windows 8 0.0.0): Executed 1 of 1 SUCCESS (0.053 secs / 0.002 secs)
TOTAL: 3 SUCCESS
```

Рис. 20.4. Запуск исполнителя тестов Карма в командной строке

При использовании Карма вы заодно пользуетесь преимуществами непрерывного тестирования. Карма также отслеживает изменения во всех исходных и тестовых файлах и перезапускает тесты в случае необходимости, мгновенно предоставляя разработчикам обратную связь.

ПРИМЕЧАНИЕ

Карма также может интегрироваться в существующие процессы сборки на базе Gulp при помощи плагина `gulp-karma`. В нашем приложении будет использоваться простой процесс командной строки, но если вас интересует эта возможность, дополнительную информацию можно найти по адресам <https://github.com/karma-runner/gulp-karma> и <https://github.com/karma-runner/grunt-karma>.

Являясь инструментом командной строки, Карма легко встраивается в систему сборки с непрерывной интеграцией. Однако, скорее всего, вы не станете запускать Карма на своем билд-сервере с такими же настройками, как на машине разработки. Например, вы не захотите, чтобы в Карма отслеживались изменения исходных и тестовых файлов, потому что в этом случае сборка будет вечно пребывать в состоянии ожидания. Вместо этого тесты должны запускаться однократно. Также можно упростить процесс сборки на сервере и запускать тесты только для PhantomJS. Открытие окон браузеров на сервере, не имеющем сеанса пользова-

тельского интерфейса, может вызвать проблемы, а на некоторых серверах исполнитель тестов «зависает» при попытке открыть окно браузера. Большинство настроек может быть задано из командной строки.

```
karma start --single-run --browsers PhantomJS
```

Также можно выбрать генератор отчетов для используемого вами билд-сервера. Хорошая документация по интеграции билд-сервера с другими возможностями Karma доступна по адресу <https://karma-runner.github.io/>.

Другие виды тестирования

Мы достаточно подробно рассмотрели модульное тестирование C# и JavaScript, но существуют и другие виды автоматизированного тестирования. Интеграционное тестирование — еще один уровень автоматизированного тестирования, которое проверяет поведение приложения при объединении всех модулей. Иначе говоря, вместо того чтобы тестировать отдельные части в изоляции друг от друга, вы тестируете их взаимодействие с другими реальными фрагментами приложения.

В основном интеграционные тесты пишутся с использованием тех же инструментов, что и модульные тесты, но каждый тест требует более тщательной подготовки. Например, вам может понадобиться реляционная база данных с тестовой информацией. Возможно, также потребуется запустить веб-сервер для размещения приложения.

Также вам придется принять еще одно важное решение: включать ли пользовательский интерфейс в интеграционные тесты? Некоторые интеграционные тесты пытаются моделировать взаимодействие пользователя с браузером с использованием таких средств автоматизации, как Selenium WebDriver (<http://www.seleniumhq.org/projects/webdriver/>). Другие ограничиваются моделированием запросов HTTP, которые будут генерироваться браузером.

В официальной документации ASP.NET Core имеется хорошее руководство по интеграционному тестированию: <https://docs.asp.net/en/latest/testing/integration-testing.html>.

Еще один вид тестирования, который также следует принять во внимание, — нагрузочное тестирование. Хуже всего узнать о проблемах производительности тогда, когда сервер начинает сбоить под высокой нагрузкой. К счастью, существуют превосходные инструменты нагрузочного тестирования (<https://www.visualstudio.com/en-us/docs/test/performance-testing/getting-started/getting-started-with-performance-testing>). Еще один хороший вариант — Web Surge (<http://websurge.west-wind.com/>).

Итоги

Фреймворк ASP.NET Core с самого начала проектировался с расчетом на автоматизацию тестирования. Использование внедрения зависимостей через конструктор в ASP.NET Core (см. главу 14 «Внедрение зависимостей») упрощает настройку, необходимую для тестирования компонентов приложения. Для тестирования ожидаемых взаимодействий с зависимостью классу могут передаваться фиктивные объекты (макеты). В этой главе мы рассмотрели ряд конкретных примеров тестирования сервисов, контроллеров, компонентов представлений и тег-хелперов.

Типичное приложение ASP.NET Core состоит как из кода C#, так и из кода JavaScript. Модульное тестирование должно охватывать обе области. В этой главе мы использовали Jasmine и Karma для организации автоматизированного модульного тестирования клиентского кода JavaScript. С правильно выбранными инструментами и определенным уровнем дисциплины в вашей команде автоматизированное тестирование не создаст проблем. Хорошо проработанный набор автоматизированных тестов поможет вашей группе двигаться вперед с большей уверенностью, — а возможно, и с более высокой скоростью.

Методы модульного тестирования, описанные в этой главе, могут применяться и к другим аспектам ASP.NET Core Framework — в частности, к точкам расширения, о которых речь пойдет в следующей главе.

21

Расширение фреймворка

Проходя мимо рабочего места Честера, Даниэль заметила, что он читает книгу. Ее всегда интересовало, что читают другие люди, и она не могла не спросить: «Что ты читаешь, Честер?»

«Ой, — сказал Честер, застигнутый врасплох. — Это книга про ASP.NET MVC Core... называется... Довольно любопытно. Многое из того, о чем рассказывается в начальных главах, вроде бы уже встречалось нам в этом проекте. Но в главе про расширение оказалось много того, чего я не знал».

«Мы в общем-то брали то, что нам нужно в настоящий момент, и не изучали фреймворк в целом. Есть ли здесь то, что нам могло бы пригодиться?» — спросила Даниэль.

«По правде говоря, мы пользуемся уже готовыми возможностями, и они просто работают. Если бы мы делали что-то нетривиальное с другими конвейерами или писали собственное промежуточное ПО, то эта глава была бы как раз для нас. Скажем, вот это, — сказал Честер, постукивая пальцем по странице, — про изоморфные приложения».

«Изоморфные приложения? Такие бесформенные, как амеба?»

«Ха-ха, — рассмеялся Честер. — Я думаю, ты путаешь изоморфные приложения с аморфными... если они бывают. Замечала, что, когда мы загружаем страницу в одностраничном приложении, загрузка часто занимает много времени?»

«Ага», — ответила Даниэль. Ее такие приложения нередко раздражали.

«Хорошо, изоморфное приложение генерирует исходную модель DOM на сервере и передает уже готовую страницу. Тогда код JavaScript на стороне клиента берет и использует уже готовую модель DOM. Такое решение будет работать быстрее, потому что страница уже будет сгенерирована, а JavaScript ничего особенно изменять не придется».

«Такое приложение должно работать лучше и быстрее. Было бы неплохо проделать это в нашем проекте».

«Все необходимые инструменты есть, можно начать в любой момент. Конечно, мне хотелось бы убрать раздражающую начальную задержку. Веб-страница не должна начинать с картинки “Подождите, пожалуйста...”, это просто глупо».

«Да уж, — сказала Даниэль. — С таким же успехом можно было сделать экран-заставку. И я думаю, с добавлением JavaScript загрузка страницы будет занимать все больше времени. Такое решение плохо масштабируется».

Фреймворк ASP.NET Core MVC создается с расчетом на расширение. Почти на каждом уровне существуют точки расширения для поддержки нетривиальных сценариев. Некоторые из этих точек уже упоминались ранее, а в этой главе мы рассмотрим их чуть подробнее.

Соглашения

Фреймворк ASP.NET Core MVC в значительной мере основан на соглашениях. По соглашению представление находится в файле `Views/ИмяКонтроллера/Имя-Действия.cshtml`. Контроллеры по соглашению находятся в папке `Controllers`. Эти соглашения избавляют разработчика от части усилий, необходимых для работы приложения. Принцип «соглашения прежде конфигурации» позволяет быстро добавлять функции и расширять приложение ASP.NET Core MVC с минимальными усилиями.

Соглашения фреймворка по умолчанию обычно хорошо работают во многих приложениях, но возможны ситуации, в которых вам хотелось бы изменять существующие и даже добавлять собственные нестандартные соглашения.

В табл. 21.1 перечислены типы соглашений, которые могут изменяться или создаваться в ASP.NET Core MVC.

Таблица 21.1. Типы соглашений в ASP.NET Core MVC

Соглашение	Интерфейс	Описание
Приложение	<code>IApplicationModelConvention</code>	Предоставляет доступ к соглашениям уровня приложения, с возможностью перебора всех нижележащих уровней
Контроллер	<code>IControllerModelConvention</code>	Соглашения, относящиеся к контроллеру, а также к более низким уровням
Действие	<code>IActionModelConvention</code>	Здесь вносятся изменения в соглашениях уровня действий, а также уровня параметров действий
Параметр	<code>IParameterModelConvention</code>	Только для параметров

При запуске приложение ASP.NET Core использует все добавленные вами соглашения, начиная с верхнего уровня (соглашения уровня приложения) и с постепенным переходом к соглашениям уровня контроллеров и действий, до соглашений параметров. При таком подходе самые конкретные соглашения применяются в последнюю очередь. Важно заметить, что соглашения уровня параметров, добавленные с использованием `ApiControllerModelConvention`, могут быть заменены через `IParameterModelConvention` независимо от порядка добавления соглашений в проект. В этом отношении ситуация отличается от промежуточного ПО, потому что порядок применения соглашений важен только в пределах одного уровня, и сам уровень определяет приоритет, который не может быть изменен.

Создание нестандартных соглашений

Чтобы создать нестандартное соглашение, создайте класс, наследующий от соглашения соответствующего уровня. Например, чтобы создать нестандартное соглашение, применяемое на уровне приложения, реализуйте интерфейс `IApplicationModelConvention`. Каждый из интерфейсов, перечисленных в табл. 21.1, содержит единственный метод `Apply`, применяемый во время выполнения. Метод `Apply` получает модель соответствующего уровня. Например, метод `Apply` интерфейса `IApplicationModelConvention` получает аргумент `ApplicationModel`. Среди прочего, `ApplicationModel` содержит список контроллеров, обнаруженных при запуске. В методе `Apply` можно проанализировать этот список и изменить свойства контроллеров.

Нестандартное соглашение уровня приложения

```
public class CustomApplicationModelConvention : IApplicationModelConvention
{
    public void Apply(ApplicationModel application)
    {
        // Здесь применяются нестандартные соглашения
    }
}
```

Соглашения регистрируются в фреймворке MVC в начале работы, когда мы вызываем метод расширения `AddMvc` из `Startup.ConfigureServices`. Все, что для этого необходимо сделать, — добавить экземпляр нестандартного соглашения в список `Conventions`.

Регистрация нестандартных соглашений в `Startup.Configure`

```
services.AddMvc(mvcOptions =>
{
    mvcOptions.Conventions.Add(new CustomApplicationModelConvention());
});
```

Рассмотрим простой пример с созданием нестандартной реализации `IActionModelConvention`. В рамках этого соглашения мы хотим автоматически добавить

фильтр `ValidateAntiForgeryToken` ко всем методам действий, которые имеют атрибут `HttpPost` и к которым не был явно добавлен `ValidateAntiForgeryToken`. В методе `Apply` мы можем проанализировать атрибуты метода действия и добавить фильтр только к тем методам действий, на которые распространяется наше соглашение.

Соглашение уровня действия, автоматически добавляющее фильтр

`ValidateAntiForgeryToken`

```
public class AutoValidateAntiForgeryTokenModelConvention : IActionModelConvention
{
    public void Apply(ActionModel action)
    {
        if (IsConventionApplicable(action))
        {
            action.Filters.Add(new ValidateAntiForgeryTokenAttribute());
        }
    }

    public bool IsConventionApplicable(ActionModel action)
    {
        if ( action.Attributes.Any(f => f.GetType() ==
            typeof(HttpPostAttribute)) &&
            !action.Attributes.Any(f => f.GetType() == typeof
                (ValidateAntiForgeryTokenAttribute))){
            return true;
        }
        return false;
    }
}
```

ПРИМЕЧАНИЕ

При помощи соглашений можно реализовать очень интересные сценарии. В MSDN Magazine Стив Смит (Steve Smith) описывает возможность структурирования приложений с использования соглашений¹. Хорошо осведомленный Филип Войцисшин (Filip Wojcieszyn) также приводит в своем блоге замечательные примеры применения префиксов глобальных маршрутов² и предлагает интересный подход к предоставлению локализованных маршрутов³.

Создавать нестандартные соглашения в ASP.NET Core MVC несложно. Конечно, при внесении изменений в соглашения ASP.NET Core MVC по умолчанию необходимо действовать осторожно. При любых изменениях в соглашениях по умолчанию разработчику труднее понять приложение, потому что оно не похоже на другие приложения ASP.NET Core MVC.

¹ <https://msdn.microsoft.com/magazine/mt763233>.

² <http://www.strathweb.com/2016/06/global-route-preix-with-asp-net-core-mvc-revisited/>.

³ <http://www.strathweb.com/2015/11/localized-routes-with-asp-net-5-and-mvc-6/>.

Промежуточное ПО

Как было показано в главе 3 «Модели, представления и контроллеры», приложения ASP.NET Core представляют собой серию компонентов промежуточного ПО, которые обрабатывают входящие запросы и манипулируют ответами. С точки зрения архитектуры в ASP.NET Core используется хорошо известный паттерн «Каналы и фильтры», который более подробно объясняется в отдельной статье MSDN по адресу <https://msdn.microsoft.com/en-us/library/dn568100.aspx>. Одно из важнейших преимуществ такой архитектуры — простота расширения. Промежуточное ПО легко добавляется в приложение, не требуя крупномасштабных изменений в других модулях.

Настройка конвейера

В приложении ASP.NET Core конвейер настраивается в методе `Startup.Configure`. В него обычно включается некоторое количество встроенных компонентов промежуточного ПО, которые легко добавляются вызовом соответствующего метода расширения `IApplicationBuilder`. В следующем примере кода приведена упрощенная версия конвейера, настроенного в приложении `AlpineSkiHouse.Web` (рис. 21.1).

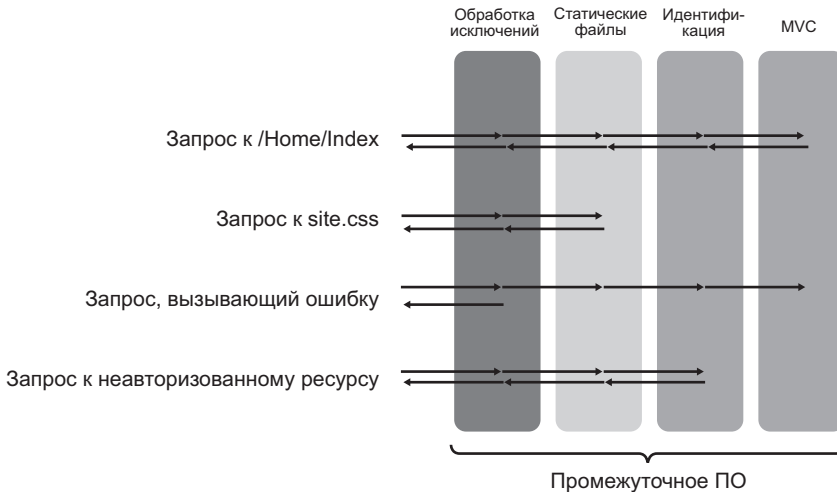


Рис. 21.1. Упрощенный конвейер запросов в `AlpineSkiHouse.Web`

Сокращенная версия метода `Startup.Configure` из `AlpineSkiHouse.Web`

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseExceptionHandler("/Home/Error");
    app.UseStaticFiles();
    app.UseIdentity();
}
```

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
}
```

Элегантность концепции промежуточного ПО в ASP.NET Core проявляется в простоте внесения изменений в конвейер запросов. Отличный пример такого рода — включение специальных страниц обработки ошибок и исключений только при выполнении приложения в среде разработки.

Пример настройки промежуточного ПО для сред разработки

```
if (env.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
    app.UseDatabaseErrorPage();
}
else
{
    app.UseExceptionHandler("/Home/Error");
}
```

Всего с одним изменением конвейера вы получаете доступ к специальным страницам диагностики ошибок в локальной среде разработки. Пример страницы ошибки базы данных показан на рис. 21.2.

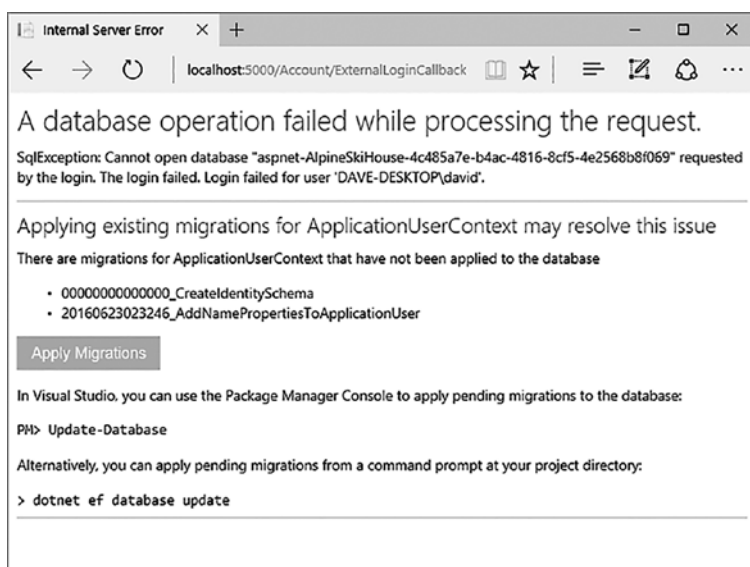


Рис. 21.2. Страница ошибки базы данных в приложении AlpineSkiHouse.Web

Помимо использования встроенных компонентов промежуточного ПО, вы также легко можете писать собственные компоненты или подгружать их через NuGet. Нестандартное промежуточное ПО добавляется в конвейер вызовом метода `Use` интерфейса `IApplicationBuilder`.

Написание промежуточного ПО

В элементарных приложениях нет необходимости реализовать нестандартное промежуточное ПО, потому что для большинства основных сценариев использования обычно уже существуют встроенные или сторонние компоненты. Однако полезно понимать, как реализовано промежуточное ПО, хотя бы для того, чтобы эта концепция не казалась такой таинственной.

Попросту говоря, промежуточное ПО представляет собой делегата, который выполняет действия с запросом, а затем передает запрос следующему делегату в конвейере.

Рассмотрим простой пример промежуточного ПО, добавляющего в ответ заголовок `Content-Security-Policy`. Такое поведение реализуется объявлением анонимного метода в методе `Startup.Configure`.

ПРИМЕЧАНИЕ

Заголовок `Content-Security-Policy` — важный инструмент защиты сайта от межсайтовых сценарных атак. О том, как работает этот заголовок и почему он важен, рассказано в соответствующем эпизоде ASP.NET Monsters на канале Channel 9¹.

Пример реализации промежуточного ПО в анонимном методе

```
app.Use(async (context, next) =>
{
    context.Response.Headers.Add("Content-Security-Policy", "default-src 'self'");
    await next.Invoke();
});
```

Впрочем, анонимные методы создают проблемы с тестированием или повторным использованием. Обычно лучше реализовать промежуточное ПО в виде класса. Класс промежуточного ПО реализует асинхронный метод `Invoke` и получает `RequestDelegate` посредством внедрения зависимости через конструктор. Обратите внимание: классы промежуточного ПО участвуют во внедрении зависимостей так же, как любые другие сервисы в ASP.NET Core. После того как класс будет создан, промежуточное ПО настраивается в `Startup.Configure`.

¹ <https://channel9.msdn.com/Series/aspnetmonsters/ASP-NET-Monsters-66-Content-Security-Policy-Headers>.

Промежуточное ПО для добавления заголовка Content-Security-Policy

```
public class CSPMiddleware
{
    private readonly RequestDelegate _next;

    public CSPMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task Invoke(HttpContext context)
    {
        context.Response.Headers.Add("Content-Security-Policy",
            "default-src 'self'");
        await _next.Invoke(context);
    }
}
```

Настройка CSPMiddleware в Startup.Configure

```
app.UseMiddleware<CSPMiddleware>();
```

После создания и реализации промежуточного ПО в любой запрос к веб-сайту Alpine Ski House включается заголовок Content-Security-Policy, как показано на рис. 21.3.

Headers	Body	Parameters	Cookies	Timings
Request URL: http://localhost:5000/				
Request Method: GET				
Status Code: 200 / OK				
Request Headers				
Response Headers				
Content-Encoding: gzip				
Content-Security-Policy: default-src 'self'				
Content-Type: text/html; charset=utf-8				
Date: Thu, 13 Oct 2016 02:33:12 GMT				
Server: Kestrel				
Transfer-Encoding: chunked				
Vary: Accept-Encoding				
X-Powered-By: ASP.NET				
X-SourceFiles: =?UTF-8?B?QzpcZGV2XG1vbN0ZXJzXEFs...				

Рис. 21.3. Заголовок ответа Content-Security-Policy

Конечно, это очень простой пример. Также возможно добавить после вызова `await _next.Invoke(context)` логику, выполняемую после исполнения всех компонентов промежуточного ПО. Любой запрос в конвейере также имеет возможность не активизировать следующий компонент промежуточного ПО. Этот прием называется замыканием конвейера запросов. Пример замыкания запроса к `site.css` встречается на рис. 21.1.

ПРИМЕЧАНИЕ

В предыдущих версиях ASP.NET для реализации нестандартной логики обработки запросов использовались модули и обработчики. Обработчики использовались для обработки запросов с заданным именем или расширением файла, а модули активизировались для каждого запроса и имели возможность замыкать обработку запросов или модифицировать ответ HTTP. Модули и обработчики использовались для управления конвейером запросов в старых версиях ASP.NET; в ASP.NET Core они были заменены промежуточным ПО. За информацией о том, как перейти с обработчиков и модулей на промежуточное ПО, обращайтесь к официальной документации ASP.NET Core по адресу <https://docs.asp.net/en/latest/migration/http-modules.html>.

Разветвление конвейера

Промежуточное ПО ASP.NET Core можно настроить таким образом, чтобы конвейер разветвлялся в зависимости от выбранных вами критериев. Простейший вариант ветвления основан на пути запроса с использованием метода расширения `Map` реализации `IApplicationBuilder`. В следующем примере любые запросы, начинающиеся с пути `/download`, используют `CustomFileDownloadMiddleware` вместо промежуточного ПО, настроенного для обычного конвейера MVC. Разветвленный конвейер изображен на рис. 21.4.

Пример разветвления конвейера с использованием `Map`

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseExceptionHandler("/Home/Error");
    app.Map("/download", appBuilder =>
    {
        appBuilder.UseMiddleware<CustomFileDownloadMiddleware>();
    });
    app.UseStaticFiles();
    app.UseIdentity();
    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

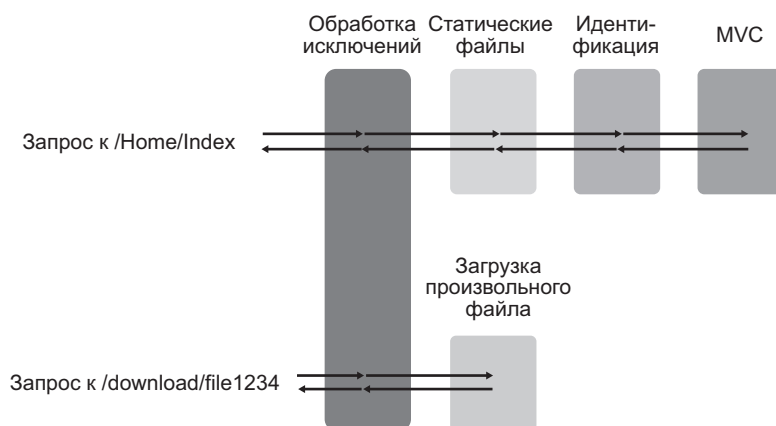


Рис. 21.4. Простой пример разветвления конвейера

Более сложные стратегии разветвления могут реализоваться с использованием функции `MapWhen`, которая позволяет реализовать логику на основании предиката типа `Func<HttpContext, bool>`. Это означает, что функция может использоваться для анализа текущего объекта `HttpContext` и принятия решения о том, нужно ли выполнять переход, на основании текущего запроса.

Загрузка внешних контроллеров и представлений

По умолчанию ASP.NET Core MVC обнаруживает контроллеры из всех сборок, ссылки на которые присутствуют в приложении. Все контроллеры можно разместить в отдельном проекте с именем `AlpineSkiHouse.Web.Controllers`, и если `AlpineSkiHouse.Web` содержит ссылку на проект `AlpineSkiHouse.Web.Controllers`, все будет работать так, как положено. В случае `AlpineSkiHouse.Web` выделение контроллеров в отдельный проект никакой пользы не приносит и только оборачивается напрасным усложнением решения. Тем не менее загрузка контроллеров и других компонентов приложений из внешних сборок может приносить пользу в некоторых ситуациях. Например, полноценная система управления контентом с большой вероятностью будет предоставлять возможность расширения, основанную на использовании плагинов.

Хотя контроллеры автоматически обнаруживаются для проектов с прямыми ссылками, следует учитывать пару обстоятельств. Во-первых, представления Razor не обнаруживаются автоматически из внешних проектов. Во-вторых, автоматическое обнаружение контроллеров не поддерживается для сборок, загружаемых во время выполнения. К счастью, ASP.NET Core MVC предоставляет точки расширения для поддержки обоих сценариев.

Загрузка представлений из внешних проектов

Чтобы загрузить представления из внешнего проекта, сначала необходимо сохранить эти представления прямо в сборке в виде встроенных ресурсов. Если предположить, что все представления хранятся в папке **Views**, то вы можете указать, что файлы **cshtml** из папки **Views** должны быть встроены в сборку, добавив в файл **project.json** проекта следующую строку:

```
"buildOptions": {  
  "embed": "Views/**/*.cshtml"  
}
```

Затем следует добавить в основной проект ссылку на пакет **NuGet Microsoft.Extensions.FileProviders** и настроить **Razor** для поиска представлений среди встроенных файлов указанного проекта.

Настройка **Razor** для загрузки представлений из встроенных ресурсов

```
services.AddMvc();  
  
// Получение ссылки на сборку со встроенными представлениями  
var assembly = typeof(AlpineSkiHouse.Web.Controllers.ExternalController).  
GetTypeInfo().Assembly;  
  
// Создание объекта EmbeddedFileProvider для этой сборки  
var embeddedFileProvider = new EmbeddedFileProvider(  
    assembly  
);  
  
// Включение провайдера в механизм представлений Razor  
services.Configure<RazorViewEngineOptions>(options =>  
{  
    options.FileProviders.Add(embeddedFileProvider);  
});
```

После этого небольшого изменения представления и контроллеры будут загружаться из внешнего проекта.

Загрузка контроллеров из внешних сборок

В структуре с модульным расширением внешние проекты вряд ли будут известны во время компиляции. Вместо этого подключаемые сборки будут загружаться во время выполнения, а динамически загружаемые сборки должны явно добавляться к **IMvcBuilder** вызовом метода **AddApplicationPart**.

Динамическая загрузка сборки и ее добавление к **IMvcBuilder**

```
var assembly = AssemblyLoadContext.Default.LoadFromAssemblyPath  
    ("fullpathtoassembly.dll");  
  
services.AddMvc()  
    .AddApplicationPart(assembly);
```

Объединение этих двух подходов позволяет относительно легко создать специализированную архитектуру с плагинами в приложениях ASP.NET Core MVC. Конечно, полноценная архитектура с плагинами требует существенно более серьезного планирования в отношении безопасности приложения, настройки динамически загружаемых сборок и обработки ошибок.

Маршрутизация

Маршрутизацией (routing) в ASP.NET Core называется процесс установления соответствия между URL-адресом входного запроса и обработчиком маршрута. Когда речь заходит о маршрутизации в ASP.NET Core, обычно имеется в виду маршрутизация MVC, которая связывает входной запрос с конкретным методом действия контроллера.

Маршрутизация в MVC настраивается определением маршрутов для `IRouteBuilder` при вызове метода `UseMvc` в `Startup.Configure`. Типичное приложение ASP.NET Core MVC использует поведение маршрутизации по умолчанию, которое задается шаблоном `File > New Project` в Visual Studio.

Конфигурация маршрутизации по умолчанию в приложении `AlpineSkiHouse`

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

Маршруты создаются вызовом `MapRoute` для `IRouteBuilder`. Каждый маршрут состоит из имени и шаблона. Промежуточное ПО маршрутизации использует специальный шаблон для установления соответствия между URL-адресом входного запроса и конкретным маршрутом. Шаблон состоит из нескольких параметров-маршрутов, каждый из которых заключается в фигурные скобки. Маршрут по умолчанию, упоминавшийся ранее, состоит из трех параметров: `{controller}`, `{action}` и `{id}`. Для параметров `controller` и `action` были заданы значения по умолчанию.

Например, если путь не включает секцию, которая соответствует части `action` шаблона, используется действие `Index`. Аналогичным образом, если также не задан параметр `controller`, используется контроллер `Home`. Часть `id` тоже не является обязательной, но для нее значение по умолчанию не задано. Если у `id` есть соответствие в URL, то значение `id` связывается с аргументом `id` метода действия. В табл. 21.2 приведены примеры URL-адресов и пар «контроллер/метод действия» для этих адресов.

Таблица 21.2. Примеры URL-адресов с соответствиями контроллеров и методов действия

URL-адрес запроса	Контроллер	Метод действия
Home/Index/	HomeController	Index()
Home/	HomeController	Index()
/	HomeController	Index()
SkiCard/	SkiCardController	Index()
SkiCard/Create/	SkiCardController	Create()
SkiCard/Edit/1	SkiCardController	Edit(int id) с id = 1

Маршрут по умолчанию — пример маршрутизации на основе соглашений. Такая маршрутизация позволяет добавлять новые контроллеры и методы действий, не задумываясь о сопоставлении новых URL. Также при маршрутизации на основе соглашений можно определить несколько разных маршрутов.

Пример с сопоставлением нескольких маршрутов

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "cards",
        template: "Cards/{action=Index}/{id?}",
        defaults: new { controller = "SkiCard" });
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

В этом примере определяется маршрут, который сопоставляет запросы, начинающиеся с *Cards*, со *SkiCardcontroller*. Использование нескольких маршрутов помогает определить более конкретные или удобные для пользователя URL, но за него придется расплачиваться возрастанием сложности. Чем больше маршрутов вы определяете, тем сложнее разобраться в том, как URL запросов сопоставляются с контроллерами и методами действий. Маршруты обрабатываются в порядке их добавления. Все входные запросы сначала проверяются по маршруту *Cards*. Если URL не удастся обработать по этому маршруту, то используется маршрут по умолчанию.

Маршрутизация на основе атрибутов

Также существует маршрутизация на основе атрибутов. В этом случае шаблоны маршрутов определяются прямо для контроллеров и методов действий при помощи атрибута *Route*.

Простой пример маршрутизации на основе атрибутов

```
public class HomeController : Controller
{
    [Route("/")]
    public IActionResult Index()
    {
        return View();
    }

    [Route("/about")]
    public IActionResult About()
    {
        return View();
    }

    [Route("/contact")]
    public IActionResult Contact()
    {
        return View();
    }
}
```

`HomeController` в этом примере использует атрибут `Route` для каждого метода действия, чтобы перенаправлять входные запросы к конкретным методам. Например, URL `/contact` сопоставляется с методом действия `Contact`.

Расширенная маршрутизация

Механизм маршрутизации в ASP.NET Core MVC чрезвычайно гибок. За дополнительной информацией о расширенном использовании маршрутизации (как на основе соглашений, так и на основе атрибутов) обращайтесь к официальной документации ASP.NET Core по адресу <https://docs.asp.net/en/latest/mvc/controllers/routing.html>.

Промежуточное ПО маршрутизации также может использоваться на более общем уровне, вне контекста фреймворка MVC. Настройка промежуточного ПО маршрутизации за пределами MVC позволяет сопоставлять URL-адреса с обработчиком маршрутов, который напрямую обрабатывает запросы HTTP. Полное описание расширенной маршрутизации в ASP.NET Core приведено в официальной документации ASP.NET Core по адресу <https://docs.asp.net/en/latest/fundamentals/routing.html>.

Инструменты dotnet

Инструментарий командной строки для `dotnet` — еще одна превосходная точка расширения во фреймворке. Помимо встроенной команды для `dotnet`, инструментарий также может подключаться через NuGet на уровне проекта. В главе 10

«Entity Framework Core» был описан инструмент командной строки `dotnet ef`, упрощающий процесс создания и выполнения сценариев миграции баз данных. Инструмент `dotnet ef` в действительности подключается к проекту добавлением ссылки на `Microsoft.EntityFrameworkCore.Tools` в секцию `tools` файла `project.json`. Это позволяет очень легко подключать дополнительные инструменты, предоставляемые Microsoft или другими сторонами. Так как инструментарий устанавливается локально программой NuGet, мы можем быть уверены в том, что разработчики имеют доступ к одному набору инструментов.

Добавление инструментов командной строки `dotnet` через NuGet

```
"tools": {  
  "Microsoft.EntityFrameworkCore.Tools": "1.0.0-preview2-final"  
}
```

Также возможно создать ваш собственный инструментарий `dotnet CLI`. Полный обзор возможностей расширения `dotnet CLI` доступен по адресу <https://docs.microsoft.com/en-us/dot-net/articles/core/tools/extensibility>.

Изоморфные приложения и сервисы JavaScript

Вы уже видели примеры превосходной интеграции ASP.NET Core с Node.js. Инструментарий Visual Studio содержит встроенную поддержку npm и других менеджеров пакетов, вследствие чего разработчики ASP.NET Core получают доступ к богатой экосистеме Node.js — по крайней мере для разработки на стороне клиента. Единственная область, в которой пока существуют пробелы, — исполнение JavaScript на сервере в процессе запуска приложения.

Признаем: идея выполнения JavaScript на сервере воспринимается естественно в контексте приложения Node.js, но, пожалуй, выглядит несколько странно в контексте приложения .NET. Стоит сделать шаг назад и поговорить о том, для чего это вообще может понадобиться; начнем с концепции изоморфных приложений.

Изоморфные приложения

Изоморфным приложением называется приложение, код которого может выполняться как на стороне клиента, так и на сервере. Концепция изоморфных приложений стала популярной для устранения часто упоминаемых недостатков одностраничных приложений (SPA). Когда клиент загружает исходную страницу в одностраничном приложении, HTML по сути представляет собой пустую оболочку со ссылками на сценарии. Пустая страница отображается на стороне клиента до того момента, когда будут загружены и выполнены дополнительные файлы JavaScript. Как правило, загрузка данных требует еще нескольких запросов. Такая архитектура может привести к замедлению загрузки начальной страницы и, воз-

можно, к ухудшению пользовательского опыта взаимодействия. Трудно создать одностраничное приложение, в котором перерисовка страницы не сопровождается бы мерцанием или в котором во время отработки запросов на загрузку сценариев и данных на экране не крутились бы индикаторы ожидания.

Помимо опыта взаимодействия, запрос, возвращающий пустую оболочку HTML, также может сыграть отрицательную роль в поисковой оптимизации (SEO, Search Engine Optimization). В идеале запрос от робота поисковой системы содержит разметку HTML, которая в конечном итоге отображается в клиенте.

Как проблема взаимодействия, так и проблемы SEO могут быть решены предварительной генерацией HTML на стороне сервера. На сервере появляется возможность кэширования вывода страницы, что существенно ускоряет последующие запросы.

Пакет `Microsoft.AspNetCore.SpaServices` обеспечивает возможность предварительной генерации компонентов SPA на стороне сервера. Он используется в сочетании с вашим любимым фреймворком SPA для генерации HTML на стороне сервера и последующей передачи результатов на сторону клиента, где и продолжается выполнение. Вместо привязки к конкретному фреймворку SPA (например, Angular или React) `SpaServices` предоставляет программный интерфейс для ASP.NET Core, который знает, как вызвать функцию JavaScript для предварительной генерации на стороне сервера и внедрить результат в разметку HTML, передаваемую клиенту. Метод предварительной генерации зависит от конкретного фреймворка SPA. Например, Angular 2 использует пакет `angular/universal`¹ для поддержки генерации страниц на стороне сервера.

За дополнительной информацией о пакете `SpaServices` и примерами использования Angular 2 и React обращайтесь по адресу <https://github.com/aspnet/JavaScriptServices/tree/dev/src/Microsoft.AspNetCore.SpaServices>.

Использование Node

Кроме превосходной поддержки изоморфных приложений, ASP.NET Core также предоставляет поддержку выполнения кода Node.js во время выполнения приложений ASP.NET Core. Хотя обычно для реализации функциональности удобнее использовать код .NET и пакеты NuGet, в некоторых случаях для решения конкретных задач удобнее использовать пакеты Node.js. Например, некоторые сторонние сервисы не предоставляют API для .NET, но предоставляют API для Node.js. Возможность выполнения пакетов Node.js во время выполнения открывает новые возможности расширения для приложений ASP.NET Core. Кроме пакетов, доступных в экосистеме .NET, разработчики ASP.NET Core получают доступ к огромному количеству пакетов, доступных в сообществе Node.js.

¹ <https://github.com/angular/universal>.

Функциональность использования Node.js во время выполнения приложений предоставляется пакетом `Microsoft.AspNetCore.NodeServices`. За дополнительной информацией о пакете `NodeServices`, включая примеры и распространенные сценарии использования, обращайтесь по адресу <https://github.com/aspnet/JavaScriptServices/tree/dev/src/Microsoft.AspNetCore.NodeServices>.

Итоги

Архитектура MVC всегда отличалась хорошими возможностями расширения, но архитектурные изменения в ASP.NET Core MVC подняли эти возможности на совершенно новый уровень. Введение промежуточного ПО представляет наибольшее изменение по сравнению с предыдущими версиями, а конвейер запросов становится более понятным, удобным в настройке и расширении практически в любом мыслимом сценарии.

Кроме промежуточного ПО, ASP.NET Core MVC предоставляет интерфейсы для настройки и расширения соглашений фреймворка. Маршрутизация может использоваться для обеспечения сложной обработки запросов в приложениях. Функции работы с JavaScript помогают заполнить пробел между .NET и экосистемой JavaScript/Node.js. Даже инструментарий командной строки `dotnet` предоставляет возможности расширения для создания кроссплатформенных средств командной строки. Будет непросто найти сценарий, который не удалось бы реализовать некоторой комбинацией точек расширения, встроенных во фреймворк ASP.NET Core MVC.

В главе 22 «Интернационализация» рассматриваются возможности расширения ASP.NET Core для построения приложений, поддерживающих языки и региональные стандарты разных стран мира.

22

Интернационализация

«...И поэтому в этих местах столько франкоговорящих людей». — Даниэль объясняла Балашу историю региона, в котором находился курорт Alpine Ski House.

«Никогда бы не подумал, — сказал Балаш, — что столько людей переезжает из Франции только потому, что в местных лесах кто-то когда-то нашел пару трюфелей».

«Не просто пару — в этих лесах их полно. И насколько я понимаю, это очень качественные трюфели. Вообще-то мы экспортируем их во Францию, где они продаются как местные. Потребители не различают их на вкус в готовых блюдах. А ты знал, что стоимость трюфелей доходит до \$3000 за фунт?»

«Невероятно. С другой стороны, в моем родном городке на икру тратят не меньше, поэтому не мне их судить. Значит, многие посетители говорят на французском?»

«Да, очень многие. Обычно мы нанимаем двуязычных инструкторов, которые могут проводить уроки как на французском, так и на английском».

«И спрос достаточно велик? Разве здесь большинство людей не говорят на английском?»

«Говорят, но у них есть родственники из Франции или Квебека, которые не знают языка. Несколько лет назад мы проводили исследования, и поддержка обоих языков безусловно окупалась».

«И ты думаешь, что на сайте Alpine Ski House стоит поддерживать как французский, так и английский?»

«Хмм, я об этом никогда не думала. Надо обсудить с маркетологами, у них есть статистика по нашему целевому рынку. Кажется, в прошлом году они проводили рассылки на французском. Пойдем, я познакомлю тебя с людьми, которые в этом разбираются лучше меня».

«Для начала скажи, а это вообще возможно — добавить другой язык в приложение?»

«Технически это несложно, потому что приложение относительно небольшое. Нужно выделить все существующие строки в файл ресурсов и воспользоваться менеджером ресурсов или чем-то в этом роде для загрузки строк в приложении. Я этим не занималась уже несколько лет, поэтому придется вспоминать, как это

делается. Сложности могут возникнуть с переводом — обычно на это уходит несколько недель. Придется запускать сайт на английском и добавлять другие языки, когда мы решим, что они действительно нужны».

«Спасибо, Даниэль, — сказал Балаш. — Это надо обдумать. Добавлю поддержку новых языков в планы после обеда».

Во многих общедоступных веб-приложениях бывает важно предоставить версию приложения на других языках. На данный момент сайт Alpine Ski House поддерживает только английский язык. Веб-приложение, ориентированное на самую широкую аудиторию, должно быть рассчитано на поддержку максимально возможного количества локалей. Локаль можно рассматривать как комбинацию региональных стандартов и языка. Эта комбинация играет важную роль по нескольким причинам. Во-первых, язык может зависеть от региона — например, английский язык, на котором говорят в Ирландии, несколько отличается от английского языка, на котором говорят в Канаде. Во-вторых, правила форматирования дат и чисел тоже зависят от региона. В мире хватает путаницы из-за того, что одни даты записываются в формате ММ-дд-гггг, а другие — в формате дд-ММ-гггг. Какую дату представляет запись 09-10-2016 — 10 декабря или 9 октября? Ответ зависит от региона.

Некоторые крупные веб-сайты (такие, как сайт Microsoft на рис. 22.1) могут поддерживать десятки локалей. Впрочем, многие сайты начинают с поддержки гораздо более скромного набора.

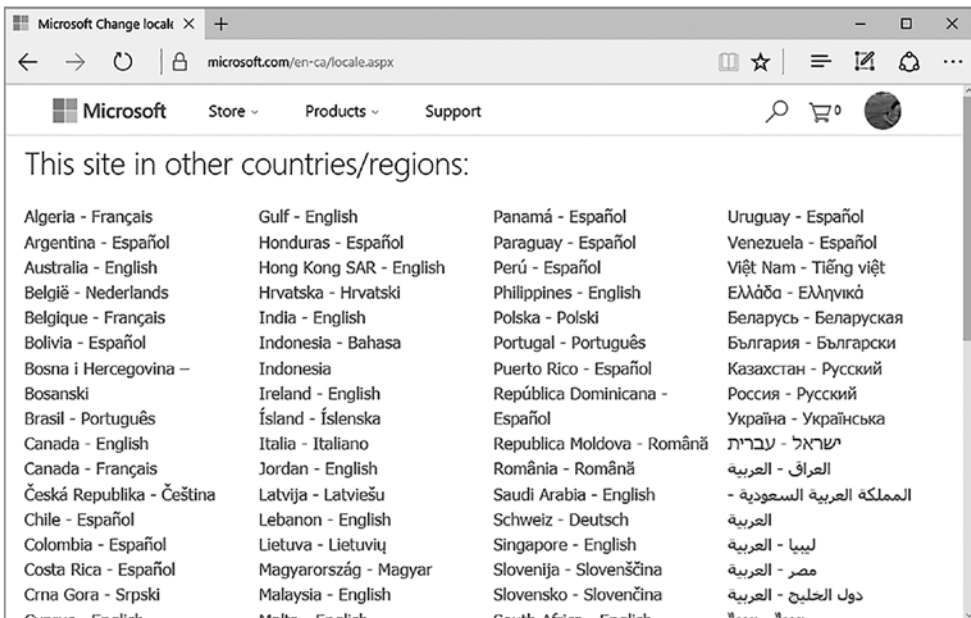


Рис. 22.1. Выбор региона и языка на сайте Microsoft

Задача поддержки многих локалей на первый взгляд выглядит устрашающе. Возможно, вас немного утешит тот факт, что ASP.NET Core предоставляет промежуточное ПО и сервисы, которые устраняют часть технических препятствий. Прежде чем погружаться в подробности реализации, для начала определимся с некоторыми важными терминами.

Локаль представляет собой комбинацию языка и региональных стандартов; также ее часто называют культурой (*culture*). Культура однозначно идентифицируется именем — двухбуквенным кодом языка, за которым следует двухбуквенный регион или код субкультуры (например, «en-CA» — английский язык в Канаде или «en-IE» — английский язык в Ирландии). Культура, которая задает язык и регион, называется *конкретной* культурой. Культура, которая задает только язык (например, «en»), называется *нейтральной*. И в завершение несколько длинных терминов, которые иногда путают. *Глобализацией* называется процесс внедрения в приложение поддержки разных культур. *Локализацией* называется процесс настройки приложения для заданной культуры. *Интернационализацией* называется процесс настройки приложения для заданного языка и региона.

В .NET всегда существовала богатая поддержка интернационализации. В .NET культура представляется классом `CultureInfo`, который среди прочего содержит информацию о форматировании дат и чисел. Каждый программный поток в .NET имеет два свойства `CultureInfo`: `CurrentCulture` и `CurrentUICulture`. Как правило, `CurrentCulture` и `CurrentUICulture` ссылаются на одну и ту же культуру. Во время выполнения свойство `CultureInfo` потока используется для определения форматов по умолчанию для дат, чисел, денежных сумм, порядка сортировки текста и других возможностей форматирования, относящихся к культуре. С другой стороны, `CurrentUICulture` используется при поиске ресурсов, относящихся к культуре, на стадии выполнения; так мы подходим к теме ресурсов.

Файл ресурса представляет собой обычный файл, содержащий пары «ключ — значение». Обычно используются текстовые значения, но значением может быть двоичный контент (например, графика). Файлы ресурсов играют важную роль, потому что они позволяют отделить локализованные строки от кода. Файлы, связанные с определенной культурой, могут загружаться во время выполнения и предоставлять переводы для выбранного языка и региона.

ГЛОБАЛИЗАЦИЯ СУЩЕСТВУЮЩИХ ПРИЛОЖЕНИЙ

Однажды мы работали над довольно крупным приложением .NET на базе WinForms, которое группа продаж собиралась вывести на международный рынок. Эти продажи обещали компании немалый доход и возможность существенного расширения клиентской базы. Однако возникла одна проблема: приложение никоим образом не было готово к локализации. Текст, который отображался в пользовательском интерфейсе, был жестко запрограммирован на всех уровнях приложения, включая базу данных.

Начался болезненный процесс выделения текста в файлы ресурсов. Один этот процесс занял у разработчиков два месяца работы. После того как строки были извлечены, мы смогли нанять переводчиков для перевода всех строк на нужный язык. Мы уже думали, что работа над проектом практически завершена... пока файл ресурсов не был загружен в приложение. В одних случаях переведенный текст не помещался на кнопках; в других надписи перекрывались с текстовыми полями, потому что макет формы был рассчитан на определенную ширину текста. Пользовательский интерфейс превратился в кашу. Нам пришлось еще несколько месяцев переделывать макеты, чтобы приложение нормально выглядело с переведенными ресурсами.

Мораль: попытка внедрить глобализацию «задним числом» в сложном готовом приложении — трудоемкий и долгий процесс. Если существует достаточно высокая вероятность того, что приложение потребует локализации, продумайте включение необходимой функциональности на ранней стадии.

Локализация текста

Чтобы обеспечить локализацию приложения, необходимо проследить за тем, чтобы весь текст, выводимый в приложении, не был жестко запрограммирован в коде приложения. Как упоминалось ранее, необходимо настроить приложение для загрузки локализованных значений из ресурсных файлов. ASP.NET Core предоставляет набор сервисных функций, с которыми этот процесс проходит относительно прямолинейно. Чтобы включить средства локализации, следует добавить локализацию в метод `Startup.ConfigureServices`.

Регистрация сервисов локализации при запуске

```
services.AddLocalization(options => options.ResourcesPath = "Resources");
services.AddMvc()
    .AddViewLocalization()
    .AddDataAnnotationsLocalization();
```

Локализация строк

Обычно большая часть локализуемого текста в приложении находится в представлении, но текст также может встречаться в контроллере или в сервисах. Например, действие `Login` контроллера `Account` добавляет текст «Invalid login attempt», который будет выведен как сообщение о неудачной проверке при визуализации представления.

Секция действия `Login` из `AccountController`

```
ModelState.AddModelError(string.Empty, "Invalid login attempt.");
return View(model);
```

Если строки жестко зафиксированы в коде, локализованная версия текста не может загружаться во время выполнения. Для решения этой проблемы ASP.NET

Core предоставляет сервис `IStringLocalizer`. Сервис `IStringLocalizer` может быть внедрен в контроллер, как и любой другой сервис.

Внедрение `IStringLocalizer` в контроллер

[Authorize]

```
public class AccountController : Controller
{
    private readonly UserManager<ApplicationUser> _userManager;
    private readonly SignInManager<ApplicationUser> _signInManager;
    private readonly IEmailSender _emailSender;
    private readonly ISmsSender _smsSender;
    private readonly ILogger _logger;
    private readonly IStringLocalizer<AccountController> _stringLocalizer;

    public AccountController(
        UserManager<ApplicationUser> userManager,
        SignInManager<ApplicationUser> signInManager,
        IEmailSender emailSender,
        ISmsSender smsSender,
        ILoggerFactory loggerFactory,
        IStringLocalizer<AccountController> stringLocalizer)
    {
        _userManager = userManager;
        _signInManager = signInManager;
        _emailSender = emailSender;
        _smsSender = smsSender;
        _logger = loggerFactory.CreateLogger<AccountController>();
        _stringLocalizer = stringLocalizer;
    }
    // Методы действий...
}
```

Теперь `AccountController` может загружать локализованные строки во время выполнения; для этого он вызывает метод `GetString` интерфейса `IStringLocalizer` и передает уникальный ключ, идентифицирующий загружаемую строку. Также существует альтернативный синтаксис с использованием индексатора по умолчанию, как делается в следующем примере кода. `IStringLocalizer` ищет локализованную версию строки для текущей культуры, используя класс `ResourceManager` фреймворка .NET. Одно из самых больших достоинств интерфейса `IStringLocalizer` заключается в том, что он не требует создания файла ресурсов для языка по умолчанию. Если уникальный ключ не найден, `IStringLocalizer` просто возвращает ключ.

Секция действия Login из AccountController с использованием IStringLocalizer

```
ModelState.AddModelError(string.Empty, _stringLocalizer["Invalid login attempt."]);
return View(model);
```


Создание файлов ресурсов

IStringLocalizer загружает файлы ресурсов на основании культуры текущего запроса, используя соглашения об именах. По этим соглашениям файлам ресурсов присваиваются имена `ContainerNamespace.CultureName.resx`, где *ContainerNamespace* — пространство имен параметра-типа, заданного в `IStringLocalizer<T>`, без пространства имен приложения по умолчанию. В случае `AlpineSkiHouse.Web.Controllers.AccountController` *ContainerNamespace* соответствует `Controllers.AccountController`. При добавлении средств локализации в приложение `AlpineSkiHouse.Web` мы указали, что ресурсы должны находиться в папке `Resources`.

Например, чтобы добавить локализованные строки для канадского французского языка, мы добавляем новый файл ресурсов с именем `Controllers.AccountController.fr-CA.resx` в папку `Resources`. Чтобы добавить файл ресурсов в Visual Studio, щелкните правой кнопкой мыши на папке `Resources` в окне `Solution Explorer` и выберите команду `Add ► New Item`. В диалоговом окне `Add New Item` выберите вариант `Code and Resource File`. Редактор файлов ресурсов в Visual Studio содержит таблицу со столбцами `Name` и `Value`. На рис. 22.2 показана запись с локализованным значением, которая представляет перевод сообщения «Invalid login attempt» на французский язык для Канады.

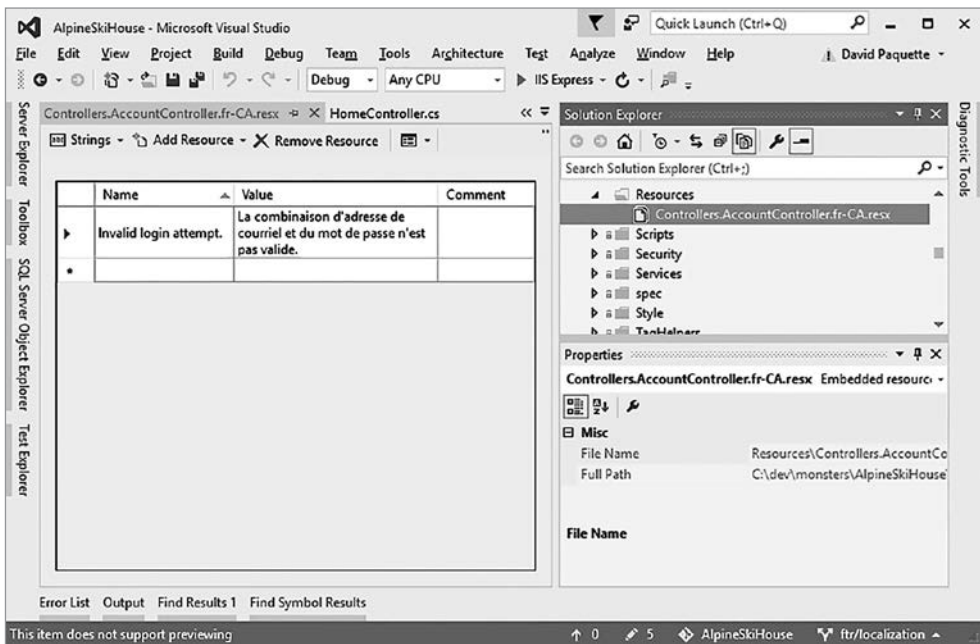


Рис. 22.2. Редактирование файлов ресурсов в Visual Studio

Файлы ресурсов можно редактировать в текстовом редакторе, потому что они содержат данные в формате XML. После схемы и информации о версии идут собственно записи данных, которые можно без труда редактировать.

Запись из файла ресурсов

```
<data name="Invalid login attempt." xml:space="preserve">
  <value>La combinaison d'adresse de courriel et du mot de passe n'est pas
    valide.</value>
</data>
```

Локализация представлений

Интерфейс `IViewLocalizer` может использоваться для предоставления локализованного текста представлению Razor. Чтобы работать с экземпляром `IViewLocalizer`, внедрите его в представление.

Использование `IViewLocalizer` в `Views/Account/Login.cshtml`

```
@using Microsoft.AspNetCore.Mvc.Localization
@model LoginViewModel
@Inject IViewLocalizer Localizer

@{
    ViewData["Title"] = Localizer["Log in"];
}

<h2>@ViewData["Title"].</h2>
<div class="row">
    <div class="col-md-8">
        <section>
            <h4>@Localizer["Use a local account to log in."]</h4>
            <!-- ... -->
        </section>
    </div>
</div>
<!-- ... -->
```

При использовании директивы `@inject` ASP.NET Core присваивает экземпляр `IViewLocalizer` локальному свойству с именем `Localizer`. Далее в представлении можно использовать `Localizer` точно так же, как мы использовали `IStringLocalizer` в контроллерах и сервисах. И снова язык по умолчанию может использоваться во внутренней реализации как уникальный ключ для локализованного текста, это означает, что для языка по умолчанию не нужно создавать файлы ресурсов. Файлы ресурсов для других культур могут быть добавлены позднее.

Реализация `IViewLocalizer` по умолчанию ищет файлы ресурсов с учетом папки текущего представления. Например, файл ресурсов с текстом для канадско-

го французского языка представления Views/Account/Login.html будет называться Resources/Views.Account.Login.fr-CA.resx.

Аннотации данных

В кодовой базе осталось еще одно последнее место, которое может содержать локализуемый текст. Аннотации данных, используемые в классах моделей представлений, позже используются тег-хелперами в Razor для отображения соответствующих меток и сообщений об ошибках проверки. Рассмотрим класс `LoginViewModel`, используемый в представлении Account/Login.

Класс `LoginViewModel`, используемый в Account/Login

```
public class LoginViewModel
{
    [Required(ErrorMessage = "Email is required")]
    [EmailAddress(ErrorMessage = "Not a valid email address")]
    [Display(Name = "Email")]
    public string Email { get; set; }

    [Required(ErrorMessage = "Password is required")]
    [DataType(DataType.Password)]
    [Display(Name = "Password")]
    public string Password { get; set; }

    [Display(Name = "Remember me?")]
    public bool RememberMe { get; set; }
}
```

Локализованные версии сообщений об ошибках аннотаций данных загружаются во время выполнения из файлов ресурсов с использованием соглашений об именах, сходных с теми, которые использует `IStringLocalizer`. Текст для канадского французского может загружаться из файла ресурсов `Resources/Models.AccountViewModels.LoginViewModel.fr-CA.resx`.

Учтите, что на момент написания книги значения `Name`, указанные в атрибуте `Display`, не локализуются. Локализация отображаемых имен будет поддерживаться в ASP.NET Core MVC 1.1 при выпуске.

Совместное использование файлов ресурсов

В примерах, рассматривавшихся до настоящего момента, каждый фрагмент приложения имел отдельный файл ресурсов. Хотя этот подход помогает разделить части приложений друг от друга, он может привести к появлению большого количества файлов ресурсов. По крайней мере часть ресурсов стоит хранить в месте, предназначенном для совместного доступа.

Сначала необходимо создать пустой класс, который будет представлять все общие ресурсы. Общие ресурсы для конкретных культур следуют соглашению об именах `Resources/SharedResources.CultureName.resx`.

SharedResources.cs

```
/// <summary>
/// Пустой класс, представляющий общие ресурсы
/// </summary>
public class SharedResources
{
}
```

Чтобы обратиться к общим ресурсам в контроллерах и сервисах, внедрите экземпляр `IStringLocalizer<SharedResources>`. Аналогичным образом также можно обращаться к общим ресурсам в представлениях, внедряя экземпляр `IHtmlLocalizer<SharedResources>`.

Использование общих ресурсов в представлении

```
@using Microsoft.AspNetCore.Mvc.Localization
@Inject IHtmlLocalizer<SharedResources> SharedLocalizer
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>@ViewData["Title"] - @SharedLocalizer["AlpineSkiHouse"]</title>
  <!-- ... -->
</head>
  <!-- ... -->
</html>
```

Теперь, когда приложение подготовлено к чтению ресурсов, относящихся к конкретной культуре, во время выполнения необходимо предоставить механизм выбора текущей культуры пользователем.

Назначение текущей культуры

В веб-приложении нельзя просто задать текущую культуру при запуске и использовать ее для всех запросов. Чтобы обеспечить поддержку пользователей из разных регионов, текущая культура должна задаваться на уровне запроса. В ASP.NET Core культура текущего запроса задается при помощи промежуточного ПО локализации.

При добавлении промежуточного ПО локализации в конвейер необходимо задать поддерживаемые культуры и культуру по умолчанию. Обратите внимание на возможность задания списка поддерживаемых культур отдельно от списка культур, поддерживаемых в пользовательском интерфейсе. Напомним, что культуры поль-

зовательского интерфейса используются для поиска ресурсов, а культуры используются для управления форматирования чисел, дат и сравнения строк. В приложении AlpineSkiHouse эти два списка совпадают.

Секция метода Startup.Configure с добавлением промежуточного ПО локализации запросов.

```
var supportedCultures = new[]
{
    new CultureInfo("en-CA"),
    new CultureInfo("fr-CA")
};

var requestLocalizationOptions = new RequestLocalizationOptions
{
    DefaultRequestCulture = new RequestCulture("en-CA"),
    SupportedCultures = supportedCultures,
    SupportedUICultures = supportedCultures
};
app.UseRequestLocalization(requestLocalizationOptions);

app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

Промежуточное ПО локализации предоставляет несколько встроенных вариантов назначения культуры каждого запроса: с использованием строк запросов, cookie или заголовков запросов. В AlpineSkiHouse используется вариант с cookie, реализованный в `CookieRequestCultureProvider`. В этом варианте текущая культура задается при помощи cookie с именем `.AspNetCore.Culture`. Если значение cookie не задано, используется культура запроса по умолчанию.

Мы должны предоставить пользователю возможность узнать, какой язык/регион выбран в настоящий момент, и выбрать поддерживаемую культуру из списка. Как и на многих веб-сайтах, для отображения текущего языка/региона будет использоваться ссылка в футере.

Секция завершителя в Views/Shared/_Layout.cshtml

```
<footer>
    @await Html.PartialAsync("_CurrentLanguage")
    <p>&copy; @DateTime.Now.Year.ToString() - AlpineSkiHouse</p>
</footer>
```

В частичном представлении `_CurrentLanguage` мы можем обратиться к текущей культуре, запросив экземпляр `IRequestCultureFeature` из контекста представления.

```
Views/Shared/_CurrentLanguage.cshtml
@using Microsoft.AspNetCore.Http.Features
@using Microsoft.AspNetCore.Localization

@{
    var requestCulture = Context.Features.Get<IRequestCultureFeature>();
}
<div class="pull-left">
    <a asp-controller="SelectLanguage"
        asp-action="Index"
        asp-route-returnUrl="@Context.Request.Path">
        @requestCulture.RequestCulture.Culture.DisplayName
    </a>
    <span> | </span>
</div>
```

Экземпляр `SelectLanguageController` может получить список поддерживаемых культур от `RequestLocalizationOptions`. Действие `Index` передает список поддерживаемых культур представлению, которое выводит эти варианты в небольшой форме. Когда пользователь выбирает один из доступных вариантов, создается объект `HttpPost` для метода действия `SetLanguage`, который добавляет cookie в ответ, задавая текущую культуру. Во всех будущих запросах текущего пользователя в качестве текущей выбирается ранее выбранная культура.

Controllers/SelectLanguageController

```
public class SelectLanguageController : Controller
{
    private readonly RequestLocalizationOptions _requestLocalizationOptions;

    public SelectLanguageController(IOptions<RequestLocalizationOptions>
requestLocalizationOptions)
    {
        _requestLocalizationOptions = requestLocalizationOptions.Value;
    }

    public IActionResult Index(string returnUrl = null)
    {
        ViewData["ReturnUrl"] = returnUrl;
        return View(_requestLocalizationOptions.SupportedUICultures);
    }

    [HttpPost]
    [ValidateAntiForgeryToken]
    public IActionResult SetLanguage(string cultureName, string returnUrl = null)
    {
        Response.Cookies.Append(
            CookieRequestCultureProvider.DefaultCookieName,
            CookieRequestCultureProvider.MakeCookieValue(new
                RequestCulture(cultureName)),
```

```

        new CookieOptions { Expires = DateTimeOffset.UtcNow.AddYears(1) }
    );

    if (returnUrl != null)
    {
        return LocalRedirect(returnUrl);
    }
    else
    {
        return RedirectToAction("Index", "Home");
    }
}
}
}

```

Метод действия `SetLanguage` использует `CookieRequestCultureProvider`, для того чтобы сгенерировать значение cookie для выбранной культуры. Для добавления cookie в `Response` используется имя cookie по умолчанию от `CookieRequestProvider` со сроком действия в 1 год.

Представление `Index` отображает простую форму и кнопку для каждой поддерживаемой культуры. Щелчок на одной из кнопок инициирует запрос `HttpPost` к методу действия `SetLanguage` с подходящим значением `cultureName`.

Views/SelectLanguage/Index.cshtml

```

@using System.Globalization
@using Microsoft.AspNetCore.Mvc.Localization

@model IEnumerable<CultureInfo>
@inject IViewLocalizer Localizer

<h2>@Localizer["Select your language"]</h2>

<div class="row">
    @foreach (var culture in Model)
    {
        <div class="col-md-3">
            <form asp-action="SetLanguage" method="post"
                asp-route-cultureName="@culture.Name"
                asp-route-returnUrl="@ViewData["ReturnUrl"]">
                <button class="btn btn-link" type="submit">
                    @culture.DisplayName
                </button>
            </form>
        </div>
    }
</div>

```

Итоги

Теперь ваше приложение полностью глобализовано, и его можно легко настроить для поддержки новых культур. Для добавления поддержки дополнительных культур потребуются файлы ресурсов для новых культур и незначительные модификации списка поддерживаемых культур в **Startup.cs**. Для всех приложений, кроме самых малых, на поздней стадии цикла разработки продукта этот процесс занимает много времени и обходится дорого. Продумайте глобализацию своего приложения как можно раньше, если существует хотя бы минимальная вероятность того, что приложению потребуется поддержка нескольких культур.

Другой важный фактор — затраты на полный перевод ресурсов приложения. Перевод занимает много времени и требует специализированных навыков; найти соответствующего специалиста может быть непросто. После того как переведенные ресурсы будут созданы, также потребуется основательное тестирование — вы должны убедиться в том, что поведение приложения соответствует ожиданиям, а весь текст был переведен правильно.

В следующей главе рассматривается некоторая усложненность кода, которая проникла в приложение Alpine Ski House; для этого мы обсудим методы рефакторинга для повышения качества кода.

23

Рефакторинг и повышение качества кода

«Уродство!» — Марк-1 явно встал на тропу войны.

«УРОДСТВО!» — он расхаживал по офису, выкрикивая и топая ногами так, как умел только он.

Даниэль сняла наушники, проверила дату на своем компьютере, вздохнула и пошла на перехват. По четвергам успокаивать Марка-1 было положено ей. Если бы сегодня была среда, то этим пришлось бы заниматься Честеру. «Что такое, Марк?»

Как и ожидалось, Марк-1 развернулся и подошел к Даниэль. «Я ненавижу этот код, Даниэль. Ненавижу каждую строку, каждую точку с запятой, каждый пробел и табуляцию... видишь, Даниэль, я беспристрастен — ненавижу их поровну».

«А что не так?»

«Да ты только посмотри на это. Методы длинные, в них встречаются неиспользуемые переменные, а это еще что? Строка с именем "Я_Болван"»?

«У нас нет такой строки», — рассмеялась Даниэль.

«Ладно, эту я придумал, зато остальные настоящие».

«Ты что-нибудь знаешь о рефакторинге, Марк?»

«Нет, это что-то связанное со старыми машинами?»

«Вовсе нет, — ответила Даниэль. — Это способ поэтапного и безопасного повышения качества кода. У меня есть книга на эту тему. Почему бы тебе не почитать ее пару часов?»

«Знаешь, Даниэль, я так и сделаю. Большое спасибо». — Умиротворенный Марк отправился к своему рабочему месту. Как только он удалился, Честер подошел к Даниэль.

«Прости, Даниэль», — смущенно сказал он.

«За что?» — спросила Даниэль.

«Я увидел, что Марк настроен воинственно, и поменял дату на твоём компьютере. Сегодня только среда».

Мало кто из разработчиков доволен кодом, который они написали год назад. Расхожая шутка на многих семинарах по программированию: программист восклицает «Кто написал этот мусор?», заглядывает в историю и выясняет, что это был он. На самом деле это замечательная возможность. Приступая к написанию кода, часто вы только пытаетесь разобраться в предметной области, задаче программирования и структуре приложения, которое вам предстоит написать. Объем информации, которую разум может усвоить за один раз, ограничен, а большое количество «подвижных частей» часто приводит к ошибкам в структуре или функциональности кода. Вернуться и исправить старый код спустя несколько месяцев, когда ваше понимание приложения улучшилось, — величайшая роскошь.

В этой главе рассматривается идея рефакторинга и возможности усовершенствований функциональности приложения без нарушения его работоспособности.

Что такое рефакторинг?

Термин «рефакторинг» был предложен в начале 1990-х годов Уильямом Опдайком (William Opdyke) и Ральфом Джонсоном (Ralph Johnson). Да, есть книга, написанная «Бандой четырех»¹. Под рефакторингом понимается изменение внутренней структуры блока кода, в результате которого код становится более понятным, удобным в тестировании, лучше адаптируется к изменениям, причем его внешняя функциональность не изменяется. Если представить себе функцию в виде «черного ящика», который получает входные данные и выдает выходные данные, изменение внутренней структуры «черного ящика» особой роли не играет, если ввод и вывод остаются неизменными (рис. 23.1).

Это наблюдение позволяет вам заменять код в приложении с сохранением функциональности и повышением общего качества приложения. И хотя мы используем термин «функция» как обозначение блока кода, его область действия не обязана ограничиваться внутренним строением функции. Вся суть объектно-ориентированного программирования — широкое введение абстракций для защиты от раскрытия подробностей реализации. Есть много конструкций, которые могут рассматриваться как «черный ящик» для целей рефакторинга. Первым кандидатом становятся классы. Реализация класса несущественна, если входные и выходные данные остаются неизменными, поэтому мы можем изменять его внутренние механизмы так, как считаем нужным. Пространства имен не обеспечивают тот же уровень защиты, что и классы, но также могут рассматриваться как кандидаты на замену. Наконец, в мире микросервисов все приложение

¹ Распространенное прозвище коллектива авторов: Эрик Гамма (Erich Gamma), Ричард Хелм (Richard Helm), Ральф Джонсон (Ralph Johnson), Джон Влиссидес (John Vlissides). — *Примеч. пер.*

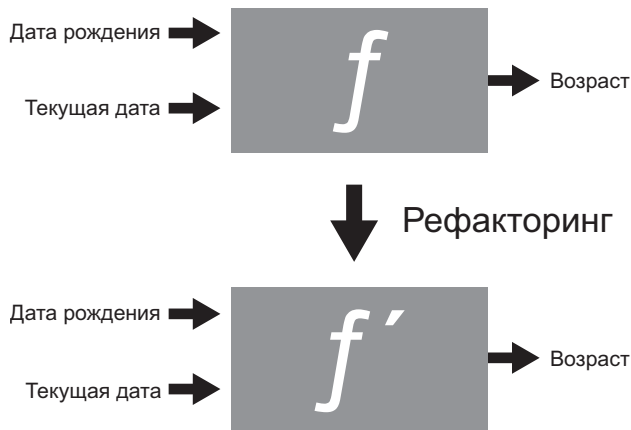


Рис. 23.1. В результате рефакторинга функция f заменяется функцией f' , которая получает те же входные данные и возвращает те же выходные данные

может быть заменено другим приложением, и такая замена тоже станет частью рефакторинга.

Рассмотрим сервис, обязанностью которого является отправка электронной почты по списку рассылки. Вводом может быть список получателей и отправляемое сообщение. Место вывода как такового здесь занимает результат: получение сообщения всеми адресатами из списка получателей (рис. 23.2).



Рис. 23.2. Рефакторинг всего микросервиса

Неважно, как именно микросервис рассылает сообщения: с помощью Sendgrid или с собственного сервера. В любом случае результат остается одним и тем же. В данном случае замена микросервиса может считаться рефакторингом.

Программисты обычно называют рефакторингом все, что способствует улучшению кода. Исключили лишний параметр метода? Добавили запись в журнал в коварном методе, чтобы обеспечить улучшенную поддержку DevOps? Хотя качество кода или качество приложения в обоих случаях повышается, ни один из них формально не может рассматриваться как рефакторинг, потому что они изменяют функциональность приложения. Полное следование определению рефакторинга требует, чтобы неизменным оставался как ввод (в том числе и состав параметров), так и вывод (включая сообщения в журнале).

Оценка качества кода

Если разработчик занимается рефакторингом для повышения качества кода, безусловно, должен существовать некоторый критерий оценки качества кода. Было бы замечательно иметь инструмент для оценки качества кода — безукоризненную метрику, способную представить все аспекты качества кода одним числом. К сожалению, такой метрики не существует. За прошедшие годы предлагалось бесчисленное множество метрик качества, от центробежного и центростремительного сцепления до оценки количества ошибок в строке кода и степени тестового покрытия кода. Выбор набора метрик качества кода — тема, которая сама по себе заслужила бы целой книги.

Впрочем, будет полезно выделить несколько простых метрик для оценки качества кода:

- **Делает ли код то, что ему положено делать?** На первый взгляд очевидно, но об этой метрике часто забывают. Ее можно оценить по количеству заявленных ошибок на тысячу строк кода или по количеству ошибок на класс. Некоторые разработчики используют эту метрику глобально, сравнивая относительное качество двух проектов по количеству обнаруженных дефектов. Впрочем, такой подход непродуктивен, потому что два проекта никогда не решают одну проблему. Можно ли ожидать, что система создания и доставки электронных поздравительных открыток будет иметь такую же степень сложности или количество ошибок, что и система управления дозатором инсулина? Нет, конечно. Эта метрика, как и большинство других, просто помогает узнать, совершенствуется ли группа и продукт со временем.
- **Удовлетворяет ли код своим нефункциональным требованиям?** Речь идет о требованиях, не связанных с вводом или выводом функции (как, например, производительность фрагмента кода или его защищенность).
- **Насколько понятен код?** Идеального кода вообще не бывает, и кому-то рано или поздно понадобится обновить его из-за обнаруженной ошибки или изменившихся требований. Этим «кем-то» может стать даже автор кода, то есть вы! Взгляните на следующий фрагмент:

```
for (var i = 1; i < parts.length; i++) {  
    root[parts[i]] = root[parts[i]] || (root[parts[i]] = {});  
    root = root[parts[i]];  
}
```

Возможно, это эффективный и умный код, но разобраться в нем почти невозможно. Найти объективную метрику понятности или удобочитаемости достаточно сложно. Эта метрика — не столько число, сколько воплощение согласованного мнения разработчиков относительно удобства сопровождения. В группах нормальных людей, не отягощенных амбициями, редко возникают серьезные расхождения по поводу того, насколько хорошо читается код. Рецензирование кода — прекрасный инструмент для обнаружения непонятных участков кода.

Другая простая, но практичная метрика — цикломатическая сложность — оценивает сложность функций по количеству ветвей кода в методе. Метод с одной командой `if` содержит две возможные ветви кода, а значит, его цикломатическая сложность равна 2 (рис. 23.3).

```
public bool CheckSkiCard(Card card)
{
    if(card.StartDate > DateTime.Now)
        return true;
    return false;
}
```

Рис. 23.3. В этой функции есть две возможные ветви кода

- **Способен ли код легко реагировать на изменения в требованиях?** И снова эта метрика не может быть выражена простым числом. Скорее это ощущение, возникающее в группе опытных разработчиков. По кислому выражению на лице разработчика, когда в его коде предполагаются серьезные изменения, можно с уверенностью определить, достигнута ли эта цель.

ВЫВОД ЦИКЛОМАТИЧЕСКОЙ СЛОЖНОСТИ В VISUAL STUDIO

Есть несколько инструментов для отображения метрик сложности кода в Visual Studio. Хорошая бесплатная программа — Code Metrics Viewer — доступна по адресу <https://visualstudiogallery.msdn.microsoft.com/03de6710-4573-460c-aded-96588572dc19>. Программа CodeRush от DevExpress заходит еще дальше и выводит рядом с каждым методом значение метрики — отличный визуальный признак, который напоминает разработчикам о том, что дальнейшее расширение функции нежелательно.

Эти четыре простые метрики станут хорошей отправной точкой для принятия решения о том, каким частям приложения может понадобиться рефакторинг. Вероятно, в процессе роста и развития приложения вам удастся обнаружить и добавить новые метрики.

Когда в вашем распоряжении оказывается несколько метрик, важно убедиться в том, что приложение не оптимизируется по ошибочной метрике. Люди склонны обманывать систему, сознательно или нет. Если выбрать метрику производительности, обеспечивающую максимально быстрый возврат из функции, но при этом не добавить метрику, устанавливающую разумное ограничение затрат памяти, появляется искушение оптимизировать функцию так, чтобы она расходовала память в обмен на вычислительные ресурсы. Подумайте, действительно ли это именно то, что вам нужно, или на самом деле конечной целью является баланс. Проанализируйте метрики и убедитесь в том, что оптимизация не идет в неверном направлении, а если все же идет — обновите набор метрик.

Выбор времени для рефакторинга

Разработчики часто жалуются, что руководство не выделяет время на проведение рефакторинга и повышение качества кода. Бизнес-сторона не видит особого смысла в повышении качества кода, пока он не создаст серьезные проблемы и не начнет приносить компании ощутимые убытки. Возможно, вы вспомните то, что говорилось выше о неправильном выборе метрик, и это, безусловно, может оказаться правдой.

Рефакторинг не обязательно проводить как отдельную операцию. Вместо этого рефакторинг должен стать естественным ответвлением работы над кодовой базой. Заметили слишком длинный метод? Переработайте его во время работы над файлом, даже если вы открыли его для других целей. Возможно, переменная с непонятным именем замедляет вашу работу во время редактирования кода. Переименуйте ее и избавьте себя от лишних трат времени при следующей работе над кодом. К рефакторингу следует относиться так же, как Роберт Баден Пауэлл (Robert Baden Powell), основатель скаутского движения, советовал мальчикам относиться к миру:

Старайтесь оставить этот мир немного лучше, чем вы его нашли, и когда придет ваше время умирать, вы можете умереть со счастливым чувством, что вы не потеряли напрасно время, но сделали самое лучшее, что смогли. “Будьте готовы” в этом направлении — жить счастливо и умереть счастливо, всегда стойко держитесь вашего Скаутского Торжественного Обещания, даже после того, как вы перестанете уже быть мальчиком, и Бог поможет вам в этом.

Роберт Баден Пауэлл

Не сохраняйте изменения, если они не повышают качество кода. В унаследованной кодовой базе исправление всего кода посредством внесения мелких усовершенствований может показаться малоэффективным, но даже небольшие изменения накапливаются со временем. Код, создающий массу проблем с сопровождением, не пишется за один день, и за один день его не исправить; но ипотечные долги годами выплачиваются небольшими ежемесячными суммами, и технологические долги возвращаются аналогичным образом. Выплата долгов, технических или финансовых, интересна тем, что даже небольшое увеличение выплат на ранней стадии может оказать огромное влияние на срок погашения долга. С уменьшением суммы основного долга сумма, идущая на погашение основного долга, а не на погашение процентов, увеличивается. Точно так же дело обстоит с техническими долгами: чем больше времени вы выделяете на выплату, тем больше времени у вас освобождается на выплату еще большей части долга. Начинайте прямо сегодня, не откладывайте!

Меры безопасности при рефакторинге

Изменение кода, работавшего годами, сопряжено с определенным риском. Даже незначительные различия в работе старого и нового кода могут создать коварные

ошибки. Прежде чем вносить изменения в существующий код, убедитесь в том, что вы их тщательно протестировали.

Если вам повезет, может оказаться, что тесты для метода уже написаны, и никакой дополнительной работы перед внесением изменений не потребуется. Впрочем, такое везение встречается редко — для унаследованных кодовых баз редко характерно хорошее покрытие автоматизированными тестами. Собственно, по мнению некоторых людей, отсутствие автоматизированных тестов — главный признак унаследованного кода.

ПРИМЕЧАНИЕ

Безусловно, самым авторитетным руководством по сопровождению и обновлению унаследованного кода является книга Майкла Физера (Michael Feather) «Working Effectively with Legacy Code». В ней представлен целый ряд замечательных решений по разрыву связей в коде и организации тестирования. Если же вам нужна книга, более тесно связанная с ASP.NET, рекомендуем книгу Саймона Тиммса (Simon Timms) и Дэвида Пакетта (David Paquette) «Evolving Legacy ASP.NET Web Applications».

Прежде чем браться за рефакторинг, необходимо написать тесты для проверки текущего поведения функции. Такие тесты, называемые *характеристическими тестами*, напрямую документируют поведение метода. Может оказаться, что тестируемый код на самом деле работает неправильно; в этом случае характеристические тесты, написанные вами, докажут его неправильность. Часто унаследованные кодовые базы зависят от неправильного поведения, так что исправление ошибки в коде нарушит работу приложения. В этом отношении характеристические тесты отличаются от большинства тестов, которые вы обычно пишете. Тестирование «неправильной» функциональности может вызвать непростые чувства у разработчиков; считайте, что эти тесты предотвращают будущие ошибки, а не устраняют существующие. Часто от неправильной функциональности зависят другие блоки кода, и исправление поведения только нарушит их работоспособность.

СОВЕТ

Технология IntelliTest разрабатывалась в Microsoft уже несколько лет; наконец, ее поддержка была включена в Visual Studio 2015 Enterprise. IntelliTest анализирует пути выполнения функций и предоставляет входные данные для выполнения всех возможных путей. Эти данные могут использоваться для создания модульных тестов, которые становятся идеальными характеристическими тестами. Впрочем, у технологии IntelliTest есть свои ограничения: она не работает с 64-разрядным кодом и не может генерировать тесты XUnit, но она способна найти граничные случаи, которые вы могли упустить из виду.

После того как метод будет обеспечен тестами, вы сможете продолжать рефакторинг, не опасаясь что-нибудь сломать.

Возьмем пример кода из приложения Alpine Ski House и создадим для него характеристические тесты, которые дадут уверенность в том, что его работо-

способность не будет нарушена при рефакторинге. Метод `Handle` из файла `AddSkiPassOnPurchaseCompleted.cs` отлично подойдет для этой цели. Это довольно длинный метод (30 строк), который в настоящее время не обеспечен никакими тестами. Он имеет довольно низкую цикломатическую сложность (3), и если не считать длины, для его рефакторинга с точки зрения сложности нет особых причин (листинг 23.1).

Листинг 23.1. Текущая версия функции-обработчика для уведомления о покупке

```
public void Handle(PurchaseCompleted notification)
{
    var newPasses = new List<Pass>();
    foreach (var passPurchase in notification.Passes)
    {
        Pass pass = new Pass
        {
            CardId = passPurchase.CardId,
            CreatedOn = DateTime.UtcNow,
            PassTypeId = passPurchase.PassTypeId
        };
        newPasses.Add(pass);
    }

    _passContext.Passes.AddRange(newPasses);
    _passContext.SaveChanges();

    foreach (var newPass in newPasses)
    {
        var passAddedEvent = new PassAdded
        {
            PassId = newPass.Id,
            PassTypeId = newPass.PassTypeId,
            CardId = newPass.CardId,
            CreatedOn = newPass.CreatedOn
        };
        _bus.Publish(passAddedEvent);
    }
}
```

На первый взгляд ввод и вывод минимальны. Метод получает объект уведомления и возвращает `void`. Но при более внимательном рассмотрении выясняется, что метод выполняет запись в базу данных и публикует события, которые тоже считаются выводом. Похоже, необходим тест для проверки того, что абонементы добавляются в базу данных, и еще один тест для проверки публикации. Также может понадобиться тест, который будет проверять, что происходит, если в коллекции `notification.Passes` нет ни одного абонемента.

Для начала создадим заготовки тестов для проверки того, как должна работать эта функция в нашем представлении (листинг 23.2).

Листинг 23.2. Заготовка тестов для обработчика уведомления о покупке

```
public class AddSkiPassOnPurchaseCompletedTests
{
    public class When_handling_purchase_completed
    {
        [Fact]
        public void Pass_is_saved_to_the_database_for_each_pass()
        {
            throw new NotImplementedException();
        }

        [Fact]
        public void PassesAddedEvents_is_published_for_each_pass()
        {
            throw new NotImplementedException();
        }

        [Fact]
        public void Empty_passes_collection_saves_nothing_to_the_database()
        {
            throw new NotImplementedException();
        }
    }
}
```

Заготовки тестов в листинге 23.2 весьма полезны: они напоминают вам о необходимости реализовать тесты. Часто во время написания одного теста вы обнаруживаете другое поведение, которое также необходимо протестировать. Вместо того чтобы переключаться на написание новых тестов, просто создайте заготовку, которая напомнит вам о необходимости вернуться к ним позже. Вероятно, больше всего сложностей будет с реализацией первого теста. Используйте контекст EF в памяти (см. главу 20 «Тестирование») и фиктивный объект. Тест в листинге 23.3 строит событие `PassPurchased` и передает его обработчику, после чего проверяются предположения относительно свойств в базе данных.

Листинг 23.3. Заполненный тест проверяет данные в контексте

```
[Fact]
public void Pass_is_saved_to_the_database_for_each_pass()
{
    using (PassContext context =
        new PassContext(InMemoryDbContextOptionsFactory.Create<PassContext>()))
    {
        var mediator = new Mock<IMediator>();
        var sut = new AddSkiPassOnPurchaseCompleted(context, mediator.Object);
        var passPurchased = new PassPurchased
        {
            CardId = 1,
            PassTypeId = 2,
```

```

        DiscountCode = "2016springpromotion",
        PricePaid = 200m
    };
    sut.Handle(new Events.PurchaseCompleted
    {
        Passes = new List<PassPurchased>
        {
            passPurchased
        }
    });

    Assert.Equal(1, context.Passes.Count());
    Assert.Equal(passPurchased.CardId, context.Passes.Single().CardId);
    Assert.Equal(passPurchased.PassTypeId, context.Passes.Single().PassTypeId);
}
}

```

Самые наблюдательные читатели могут заметить, что в листинге 23.3 тестируются два свойства созданного объекта `Pass`, тогда как в исходном коде задаются значения трех свойств. Пропущенное свойство `CreateOn` (дата создания), которому присваивается значение `UtcNow.Dates`, всегда создает проблемы с тестированием, потому что между созданием даты и ее тестированием проходит время. Часто в тесты включается хитроумная логика для проверки нечеткого соответствия дат, но в нашем решении вместо этого внедряется объект для получения дат (листинг 23.4).

Листинг 23.4. Сервис получения даты просто упаковывает значение `UtcNow` для внедрения

```

public class DateService : IDateService
{
    public DateTime Now()
    {
        return DateTime.UtcNow;
    }
}

```

Экземпляр внедряется в класс `AddSkiPassOnPurchaseCompleted` через конструктор, а в код вносятся изменения для его использования (листинг 23.5).

Листинг 23.5. Использование сервиса получения даты

```

Pass pass = new Pass
{
    CardId = passPurchase.CardId,
    CreatedOn = _dateService.Now(),
    PassTypeId = passPurchase.PassTypeId
};

```

Изменение реализации сопряжено с некоторым риском, но вы рискуете все же меньше, чем если оставите объект частично протестированным. Чтобы опреде-

лить, потребует ли изменение дополнительных тестов, руководствуйтесь здравым смыслом. Код обновленного теста приведен в листинге 23.6.

Листинг 23.6. Обновленный тест с использованием сервиса получения даты

```
[Fact]
public void Pass_is_saved_to_the_database_for_each_pass()
{
    using (PassContext context =
        new PassContext(InMemoryDbContextOptionsFactory.Create<PassContext>()))
    {
        var mediator = new Mock<IMediator>();
        var dateService = new Mock<IDateService>();
        var currentDate = DateTime.UtcNow;
        dateService.Setup(x => x.Now()).Returns(currentDate);
        var sut = new AddSkiPassOnPurchaseCompleted(context, mediator.Object,
            dateService.Object);
        var passPurchased = new PassPurchased
        {
            CardId = 1,
            PassTypeId = 2,
            DiscountCode = "2016springpromotion",
            PricePaid = 200m
        };
        sut.Handle(new Events.PurchaseCompleted
        {
            Passes = new List<PassPurchased>
            {
                passPurchased
            }
        });

        Assert.Equal(1, context.Passes.Count());
        Assert.Equal(passPurchased.CardId, context.Passes.Single().CardId);
        Assert.Equal(passPurchased.PassTypeId, context.Passes.Single().PassTypeId);
        Assert.Equal(currentDate, context.Passes.Single().CreatedOn);
    }
}
```

Этот способ выделения логики данных может пригодиться при тестировании дат — как будущих, так и прошлых. Теперь можно реализовать остальные тестовые методы (листинг 23.7).

Листинг 23.7. Код, который проверяет, что для каждого абонента публикуется событие и что его свойства правильны

```
[Fact]
public void PassesAddedEvents_is_published_for_each_pass()
{
    using (PassContext context =
```

```

        new PassContext(InMemoryDbContextOptionsFactory.Create<PassContext>()))
    {
        var mediator = new Mock<IMediator>();
        var dateService = new Mock<IDateService>();
        var currentDate = DateTime.UtcNow;
        dateService.Setup(x => x.Now()).Returns(currentDate);
        var sut = new AddSkiPassOnPurchaseCompleted(context, mediator.Object,
            dateService.Object);
        var passPurchased = new PassPurchased
        {
            CardId = 1,
            PassTypeId = 2,
            DiscountCode = "2016springpromotion",
            PricePaid = 200m
        };
        sut.Handle(new Events.PurchaseCompleted
        {
            Passes = new List<PassPurchased>
            {
                passPurchased
            }
        });
        var dbPass = context.Passes.Single();
        mediator.Verify(x => x.Publish(It.Is<PassAdded>(y => y.CardId ==
            passPurchased.CardId &&
                                                    y.CreatedOn == currentDate
&&
                                                    y.PassId == dbPass.Id &&
                                                    y.PassTypeId ==
passPurchased.PassTypeId)));
    }
}

```

Этот тест готовит обработчик и использует объект-макет, предоставленный библиотекой Moq, для проверки срабатывания событий. Лямбда-выражение, передаваемое `It.Is`, проверяет каждое свойство события.

Вероятно, вы заметили, что этот тест очень похож на предыдущий. Конечно, из тестов можно выделить общие аспекты, упрощающие построение будущих тестов. Даже в коде тестов важно стремиться к удобочитаемости. Окончательная версия теста представлена в листинге 23.8.

Листинг 23.8. Тестирование случая с передачей обработчику пустой коллекции

```

[Fact]
public void Empty_passes_collection_saves_nothing_to_the_database()
{
    using (PassContext context = GetContext())
    {
        var mediator = new Mock<IMediator>();
    }
}

```

```

var dateService = new Mock<IDateService>();
var currentDate = DateTime.UtcNow;
dateService.Setup(x => x.Now()).Returns(currentDate);
var sut = new AddSkiPassOnPurchaseCompleted(context, mediator.Object,
    dateService.Object);
sut.Handle(new Events.PurchaseCompleted { Passes = new
    List<PassPurchased>() });

Assert.Equal(0, context.Passes.Count());
}
}

```

Этот тест проверяет, что при передаче пустой коллекции в базе данных никакая информация не сохраняется. На самом деле в базе данных ничего сохраняться не должно, даже если код содержит ошибки, но существует вероятность того, что из-за пустой коллекции будут возникать исключения. Всегда желательно тестировать подобные негативные случаи.

Впрочем, при написании теста стало очевидно, что в программе отсутствует другой тест — эквивалентный тест для публикации сообщений при пустой коллекции (см. листинг 23.9).

Листинг 23.9. Тест для проверки того, что при пустой коллекции сообщения не публикуются

```

[Fact]
public void Empty_passes_collection_publishes_no_messages()
{
    using (PassContext context = GetContext())
    {
        var mediator = new Mock<IMediator>();
        var dateService = new Mock<IDateService>();
        var currentDate = DateTime.UtcNow;
        dateService.Setup(x => x.Now()).Returns(currentDate);
        var sut = new AddSkiPassOnPurchaseCompleted(context, mediator.Object,
            dateService.Object);

        sut.Handle(new Events.PurchaseCompleted { Passes = new
            List<PassPurchased>() });

        mediator.Verify(x => x.Publish(It.IsAny<PassAdded>()), Times.Never);
    }
}

```

Окончательный код теста повторяет листинг 23.8, но проверяет, что посредник не публикует сообщения.

Эти четыре теста характеризуют функцию, подвергаемую рефакторингу. Ни один из них не направлен на внутреннюю структуру функции, но эти тесты проверяют

весь ввод и вывод функции. Теперь мы можем вносить любые изменения во внутреннюю реализацию функции и не сомневаться в том, что при прохождении всех четырех тестов это не приведет к нарушению кода, зависящего от функции.

Рефакторинг без предварительных защитных мер тоже возможен, но по крайней мере вы должны чувствовать себя неуютно и беспокоиться о непредвиденных побочных эффектах изменений. А самое замечательное в тестах такого рода — то, что они превращаются в тесты общего назначения, которые укрепляют целостность вашего набора тестов. При желании характеристические тесты можно пометить аннотациями, чтобы не путать их с обычными тестами.

Разработка на основе данных

В главе 12 «Конфигурация и журналирование» говорилось о важности ведения журнала и его полезности при реальной эксплуатации приложения. Что ж, мы добрались до этой стадии. Во время первой передачи данных в группу тестирования команда Alpine Ski обнаружила, что в некоторых частях сайта выдавались исключения.

Разработчики обратились к журналам и проанализировали исключения в истории проекта, чтобы понять, в каких областях было особенно много проблем. Анализ такого рода называется *тепловым сканированием*; он используется для поиска «горячих точек» в приложении.

Если вы знаете, какие проблемы особенно часто встречаются в приложении, эта информация станет хорошей отправной точкой для начала рефакторинга. Возможно, тепловую карту ошибок стоит совместить с другой картой, которая показывает, к каким областям приложения посетители обращаются чаще всего.

Представьте, что у вас имеется статистика по количеству ошибок на запрос и ежедневному количеству запросов (табл. 23.1).

Таблица 23.1. Количество ошибок на запрос и ежедневное количество запросов

Область	Количество ошибок на запрос	Ежедневное количество запросов
Вход	0,002	123
Лендинг	0,0003	50 000
Регистрация	0,1	10
Приобретение абонемента	0,2	50

На первый взгляд кажется, что самая проблемная область — подсистема приобретения абонемента с наивысшей вероятностью ошибок, но в действительности большее количество пользователей сталкивается с ошибками на лендинге.

Многие технические организации, такие как GitLabs, Etsy и группа разработки Windows, собирают многочисленные метрики по работе приложения и взаимодействию с ним пользователей. Они не станут заниматься рефакторингом без сбора метрик для существующего кода. Характеристические модульные тесты отлично подходят для оценки функциональных свойств методов, но часто нас также интересует, не приведут ли изменения к регрессии в производительности. Возможность проанализировать реальные данные производительности в результате рефакторинга бесценна. Разработка, управляемая данными, — более масштабная тема, чем простой рефакторинг качества кода, и она безусловно заслуживает вашего внимания.

Пример чистки кода

Умение выполнять эффективную чистку кода — продолжительный процесс, на правильную реализацию которого могут потребоваться годы. Вам будет полезно рассмотреть конкретный пример кода. К счастью, как и в любом сколько-нибудь сложном проекте, в Alpine Ski House есть много кандидатов для рефакторинга. Хорошим кандидатом станет действие `Create` в классе `SkiCardController` (листинг 23.10).

Листинг 23.10. Действие `Create` в классе `SkiCardController`

```
public async Task<ActionResult> Create(CreateSkiCardViewModel viewModel)
{
    // Вернуть представление при недействительном состоянии модели
    if (!ModelState.IsValid)
        return View(viewModel);

    // Создание и сохранение карты
    string userId = _userManager.GetUserId(User);
    _logger.LogDebug("Creating ski card for " + userId);

    using (_logger.BeginScope("CreateSkiCard:" + userId))
    {
        var createImage = viewModel.CardImage != null;
        Guid? imageId = null;

        if (createImage)
        {
            _logger.LogInformation("Uploading ski card image for " + userId);
            imageId = Guid.NewGuid();
            string imageUri = await _uploadservice.UploadFileFromStream("cardimages",
imageId + ".jpg", viewModel.CardImage.OpenReadStream());
        }
    }
}
```

```

_logger.LogInformation("Saving ski card to DB for " + userId);
SkiCard s = new SkiCard
{
    ApplicationUserId = userId,
    CreatedOn = DateTime.UtcNow,
    CardHolderFirstName = viewModel.CardHolderFirstName,
    CardHolderLastName = viewModel.CardHolderLastName,
    CardHolderBirthDate = viewModel.CardHolderBirthDate.Value.Date,
    CardHolderPhoneNumber = viewModel.CardHolderPhoneNumber,
    CardImageId = imageId
};
_skiCardContext.SkiCards.Add(s);
await _skiCardContext.SaveChangesAsync();

_logger.LogInformation("Ski card created for " + userId);
}

_logger.LogDebug("Ski card for " + userId + " created successfully,
    redirecting to Index...");
return RedirectToAction(nameof(Index));
}

```

Метод выполняет проверку и сохраняет запись в базе данных. На первый взгляд метод получился довольно длинным: целых 42 строки при сложности сопровождения 206.

ПРИМЕЧАНИЕ

Сложность сопровождения — метрика, специфическая для CodeRush. Ее целью является снижение отношения «сигнал/шум» многих существующих метрик. При вычислении сложности сопровождения различным программным конструкциям (блокировкам, комментариям, блокам try/catch и т. д.) присваиваются оценки в баллах. Суммирование этих оценок дает представление о сложности методов. В идеале эти числа должны быть меньше 100. Исходное сообщение в блоге, в котором была описана логика вычислений, потерялось, но сообщение Марка Миллера (Mark Miller) было заново опубликовано по адресу http://www.skorkin.com/2010/11/code-metrics-heres-your-new-code-metric/#.V_8P__ArKUk.

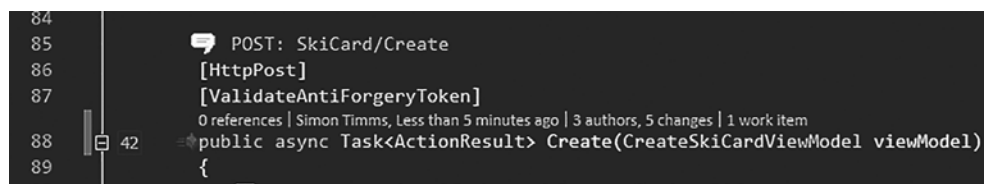


Рис. 23.4. Экран с кодом, помеченным аннотациями CodeRush. 42 слева от кода — сложность сопровождения

Прежде чем пытаться разбивать метод, чтобы сделать его более понятным и компактным, необходимо решить некоторые проблемы с качеством кода. Одно из простейших усовершенствований связано с использованием символа «+» для конкатенации строк. В современных версиях C# поддерживается интерполяция строк, которая способна заменить конкатенацию.

```
// До
_logger.LogDebug("Ski card for " + userId + " created successfully, redirecting to
                Index...");
// После
_logger.LogDebug($"Ski card for {userId} created successfully, redirecting to
                Index...");
```

Затем в программе существует переменная `imageUri`, которая создается, но в которую никогда не выполняется запись. Эту переменную можно полностью удалить, а возвращаемый результат просто игнорировать. Раз уж мы занялись переменными, присмотримся к именам переменных. Объект `SkiCard` с именем `s` выглядит невразумительно, потому что имя `s` ничего не говорит, что собой представляет переменная или что она делает. Обычно переменным, существующим только в одном экземпляре, присваивается имя, совпадающее с именем типа (листинг 23.11).

Листинг 23.11. Создание переменной `skiCard`

```
var skiCard = new SkiCard
{
    ApplicationUserId = userId,
    CreatedOn = DateTime.UtcNow,
    CardHolderFirstName = viewModel.CardHolderFirstName,
    CardHolderLastName = viewModel.CardHolderLastName,
    CardHolderBirthDate = viewModel.CardHolderBirthDate.Value.Date,
    CardHolderPhoneNumber = viewModel.CardHolderPhoneNumber,
    CardImageId = imageId
};
_skiCardContext.SkiCards.Add(skiCard);
```

Также из конструкции `var` исключается явное объявление типа переменной. Это чисто стилевое решение, но если такие языки, как F#, повсеместно могут определять типы автоматически, нет причин, по которым бы C# не мог сделать то же самое.

Далее идет проверка изображения на `null`, которая осуществляется на отдельном шаге во временной переменной с именем `createImage`. Временные переменные полезны, если вам нужно выполнить высокотратную операцию и сохранить результат для использования в нескольких местах. Похоже, к нашему примеру это не относится, потому что переменная используется всего один раз для простой

проверки на `null`, которая высокочастотной никак не является. Переменную можно просто заменить встроенной проверкой:

```
if (viewModel.CardImage != null)
{ ...
```

Метод сокращается до 40 строк, а его сложность сопровождения уменьшается до 165, и все же возможности для дальнейшего сокращения сложности еще остаются. Хорошим ориентиром для поиска фрагментов, которые можно было бы выделить из функции, является местонахождение комментариев. Споры о полезности комментариев в исходном коде еще идут, но, похоже, большинство программистов сейчас придерживается того мнения, что самодокументируемого кода вполне достаточно. Комментарии стоит применять для ответов на вопрос «почему?», а не «как?». Например, взгляните на следующий комментарий:

```
// Что это, блокировка? Тебя не волнует быстроедействие? Ты всех ненавидишь?
// Спокойно, на то есть веская причина: когда мы запрашиваем новый ключ
// у SBT, предыдущий ключ становится недействительным, поэтому при запросе
// двух ключей один из них автоматически теряется. Блокировка нужна,
// чтобы предотвратить возможную потерю наших ключей.
```

Комментарий дает некоторое представление о том, почему существует этот необычный блок кода. Комментарий в нашем примере, `// Создание и сохранение карты`, никак не помогает нам с вопросом «почему?». Он всего лишь отмечает начало блока, который было бы лучше оформить в виде функции. Мы можем выделить несколько функций из кода и удалить комментарии. В результате чистки код выглядит так, как показано в листинге 23.12.

Листинг 23.12. Код, полученный в результате рефакторинга: длинный, но более понятный и удобный в тестировании

```
private async Task<Guid> UploadImage(CreateSkiCardViewModel viewModel, string
userId)
{
    Guid imageId;
    _logger.LogInformation("Uploading ski card image for " + userId);
    imageId = Guid.NewGuid();
    await _uploadservice.UploadFileFromStream("cardimages", $"{imageId}.jpg",
viewModel.
CardImage.OpenReadStream());
    return imageId;
}
private bool HasCardImage(CreateSkiCardViewModel viewModel)
{
    return viewModel.CardImage != null;
}
private async Task CreateAndSaveCard(CreateSkiCardViewModel viewModel)
```

```

{
    var userId = _userManager.GetUserId(User);
    _logger.LogDebug($"Creating ski card for {userId}");

    using (_logger.BeginScope($"CreateSkiCard: {userId}"))
    {
        Guid? imageId = null;
        if (HasCardImage(viewModel))
        {
            imageId = await UploadImage(viewModel, userId);
        }

        _logger.LogInformation($"Saving ski card to DB for {userId}");
        var skiCard = new SkiCard
        {
            ApplicationUserId = userId,
            CreatedOn = DateTime.UtcNow,
            CardHolderFirstName = viewModel.CardHolderFirstName,
            CardHolderLastName = viewModel.CardHolderLastName,
            CardHolderBirthDate = viewModel.CardHolderBirthDate.Value.Date,
            CardHolderPhoneNumber = viewModel.CardHolderPhoneNumber,
            CardImageId = imageId
        };
        _skiCardContext.SkiCards.Add(skiCard);
        await _skiCardContext.SaveChangesAsync();

        _logger.LogInformation("Ski card created for " + userId);
    }
}

// POST: SkiCard/Create
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Create(CreateSkiCardViewModel viewModel)
{
    if (!ModelState.IsValid)
        return View(viewModel);

    await CreateAndSaveCard(viewModel);

    _logger.LogDebug($"Ski card for {_userManager.GetUserId(User)} created
        successfully, redirecting to Index...");
    return RedirectToAction(nameof(Index));
}

```

Код разбивается на функции, каждая из которых служит одной цели. Ни один метод не содержит более 30 строк, а сложность сопровождения не превышает 100. Код становится чище, а благодаря извлечению таких методов, как `HasCardImage`, он проще читается.

Полезные инструменты

Существуют замечательные программы, которые анализируют код и предлагают решения. Программа CodeRush уже упоминалась в тексте; еще один популярный инструмент — Resharper (или R#, как его часто называют). И CodeRush и R# предназначены не только для анализа, но и для рефакторинга и повышения производительности.

StyleCop и NDepend — инструменты анализа кода, которые применяют к вашему коду правила, выдерживающие хороший стиль программирования. NDepend можно настроить так, чтобы для слишком длинных или сложных методов выдавались ошибки; таким образом предотвращается само попадание плохого кода в продукт.

Самый дешевый и удобный инструмент у вас уже есть: компилятор. Простое включение режима `Warnings as Errors` в конфигурации проекта заставит вас исправлять предупреждения компилятора. Предупреждения появляются не просто так, они почти всегда указывают на некоторую неочевидную проблему в вашем коде.

На пути к качественному коду

Повышение качества кода приложения должно достигаться совместными усилиями всей команды. Самая важная составляющая повышения качества кода — не новомодный инструмент вроде CodeRush или NDepend, а чувство совместной ответственности за код в команде. Программист не должен смотреть на фрагмент кода и думать: «О, это написал Леонид, и мне это лучше не изменять». Рецензирование кода и парное программирование существенно улучшают культуру программирования в команде и избавляют участников от сильного чувства личной принадлежности кода.

ПРИМЕЧАНИЕ

При рецензировании кода очень легко выдавать негативные сентенции типа: «Зачем ты это сделал?», «По-моему, это неэффективно» и т. д. Такие комментарии никому не помогают и только разобщают команду. Лори Лалонд (Lori Lalonde) обсуждает культуру рецензирования кода в статье в своем блоге «The Code Review Blues» по адресу <http://www.westerndevs.com/the-code-review-blues/>.

Итоги

Рефакторинг и чистка кода напоминают выплату ипотечного кредита: чем быстрее вы платите, тем проще вам выплачивать остаток. Перефразируя китайскую поговорку, «Самое лучшее время для улучшения качества кода было полгода назад, второе — сейчас». Даже если каждый раз вы будете улучшать какую-нибудь

мелочь, при каждом новом сохранении с вашим кодом будет все проще и приятнее работать.

Вносить изменения в непротестированный код может быть рискованно. Лучше постройте для своего кода набор тестов, прежде чем браться за его чистку. Как правило, чистый код, который работает неправильно, хуже неряшливого, но правильно работающего кода.

Наконец, постарайтесь выработать чувство общей принадлежности кода. Небрежный, трудный в сопровождении код должен вызывать чувство отторжения у всей команды. Небрежный код обычно ведет к появлению еще более небрежного кода, тогда как чистый код стимулирует разработчиков писать больше чистого кода — проявление «теории разбитых окон» в разработке программного обеспечения¹.

В последней главе мы поговорим о том, как организовать код приложения так, чтобы он хорошо масштабировался и не создавал трудностей с сопровождением.

¹ https://ru.wikipedia.org/wiki/Теория_разбитых_окон. — *Примеч. пер.*

24

Организация кода

Все шло хорошо, и Даниэль уже начала строить планы на отпуск после завершения проекта. Ей хотелось уехать куда-нибудь, где нет склонов, абонементов, а самое главное — нет Entity Framework. На последней ретроспективе Марк жаловался на размер некоторых контроллеров. Сегодня Даниэль хорошо поняла его чувства.

Ей уже в третий раз приходилось пересматривать свои изменения при работе над контроллером. Другие разработчики вносили параллельные изменения в код, и все время опережали ее с сохранением. Даниэль была убеждена в том, что ситуацию можно улучшить, и даже была уверена в том, что знает, как это сделать. Отказавшись от всякой надежды на сохранение файла, она пригласила Марка-2 и Балаша к своему рабочему месту.

«Балаш, код выходит из-под контроля. С этим надо что-то делать», — сказала она.

«Полностью согласен», — поддержал Марк-2.

«Что ж, — сказал Балаш. — Похоже, все согласны с тем, что у нас есть технический долг, относящийся к структуре проекта. В моей родной стране есть поговорка: "Bez miki net nauki". Это означает, что испытания закаляют характер. А теперь поговорим о том, как решать эту проблему, потому что "Voda kamen tochtit"».

Даниэль не знала, что это значит, но похоже, у нее хватало политической воли для внесения необходимых изменений. А теперь нужно было как-то сохранить файл с изменениями, пока кто-нибудь снова не вмешался в дело.

Практически для всего в компьютерной области есть стандарты: для сетей, для разъемов, для языков. Есть даже стандарты по написанию стандартов (RFC 2119). Стандарты в компьютерах — огромное достижение. К примеру, вы можете купить для своего компьютера практически любую видеокарту. Вам не придется покупать видеокарту Dell для компьютера Dell, потому что интерфейс между системной платой и видеокартой определяется стандартом. Конечно, так было не всегда. На заре компьютерной эпохи для компьютеров Compaq приходилось покупать компоненты Compaq. Если бы уровень стандартизации, существующий в компьютер-

ной области, существовал бы в других областях, вы могли бы заменить деталь своей машины аналогичной деталью от другого производителя, а посудомоечные машины стоили бы дешевле, потому что их можно было бы массово выпускать по единому стандарту.

Несмотря на все сказанное, у структуры каталогов в проектах ASP.NET MVC нет формализованного стандарта. Конечно, при создании нового проекта создается некая структура, но это всего лишь рекомендация, которая может быстро превратиться в невразумительную мешанину. Вероятно, отсутствие стандартов является пережитком эпохи WebForms, когда отсутствие формализованной структуры было выигрышным моментом. Приложения WebForms напоминали приложения на языке Snowflake: из-за разных соглашений об именах и структуре папок результаты выглядели очень интересно. Собственно, с того времени мы научились ценить принцип «соглашения прежде конфигурации».

В этой главе рассматриваются некоторые недостатки текущей структуры каталогов наших проектов, а также предлагаются возможные способы ее улучшения в средних и крупных проектах.

Структура репозитория

Но прежде чем переходить к структуре проекта, следует обратиться к структуре репозитория с исходным кодом. Редко встретишь проект, в котором нет ресурсов, кроме исходного кода. Обычно в проект включаются другие составляющие, которые тоже нужно где-то хранить: логотипы, сценарии сборки, сценарии сопровождения и т. д. А значит, исходный код не стоит хранить в корневом каталоге проекта, чтобы он не смешивался с другими ресурсами. Для его хранения достаточно простого каталога `/src`.

Перемещение исходного кода вниз по дереву каталогов может вызвать недоумение у кого-то, кто только что пришел в проект, поэтому на верхнем уровне стоит разместить сценарий сборки с очевидным именем (например, `build.bat`). Сценарий может быть чем угодно: от простого пакетного файла, который вызывает файл решения в каталоге с исходным кодом, до полноценного Make-файла. Кроме того, часто бывает полезно разместить на этом уровне файл `Readme` с описанием того, что делает проект. Проходит совсем немного времени, и люди забывают, что делает тот или иной проект. `Readme`-файл хорошо подходит для размещения описаний приложений.

Состав остальных папок на этом уровне зависит от проекта. Опыт показывает, что в ходе реальной эксплуатации приложения появляется необходимость в сценариях и инструментах. Также администратору часто приходится править информацию в базе данных или манипулировать с сообщениями в очередях; такие эксплуатационные сценарии также должны где-то храниться — например, в каталоге `/tools` или `/scripts`. Такие сценарии очень полезно сохранить на будущее. Вряд ли

вы столкнетесь с точно такой же задачей, но вам вполне может встретиться что-то похожее. И тогда модификация существующих сценариев сэкономит вам много времени, особенно если эти сценарии выполняют шаблонные действия (например, правку сообщений в очереди). В хозяйстве все пригодится!

Раз уж речь зашла об эксплуатационных сценариях, в мире «инфраструктуры как кода» (IaC) должно существовать место для хранения таких сценариев. Возможность воспроизведения среды из сценариев имеет смысл только в том случае, если их версии контролируются, а история изменений отслеживается.

На этом уровне также стоит хранить документацию.

Внутри исходного кода

В начале работы появляется искушение выбрать команду **File ▶ New Project**, но в большинстве случаев с этим искушением следует бороться. Добавление новых проектов в решение замедляет сборку и только приводит к появлению большего количества DLL-библиотек. Считается, что хранение кода в разных проектах в некоторой степени способствует изоляции. Например, многие разработчики хранят интерфейсы в проектах `Core` или `Common`, а реализацию — в параллельном каталоге другого проекта (рис. 24.1). Например, каталогу `Blob` из проекта `Completions.Common` соответствует каталог `Completions.DataProject`.

Кроме того, на иллюстрации представлен проект `Completions.Common.Test` с тестами для кода, содержащегося в проекте `Completions.Common`. Как нетрудно предположить, этот подход приводит к размножению проектов в решении. На самом деле это решение содержит 22 проекта. В большинстве случаев преимущества создания новых проектов могут быть реализованы просто за счет создания пространств имен. Разделение интерфейсов и их реализаций позволяет легко добавлять новые реализации. Впрочем, в большинстве случаев реализация только одна, а разбиение файлов по разным проектам оказывается преждевременной оптимизацией. Не стоит с самого начала предполагать, что вам нужно несколько проектов, и строить соответствующую структуру приложения — лучше предполагать обратное и выделять дополнительные проекты только при необходимости.

Также на этой иллюстрации представлен полезный пример параллельной структуры каталогов.

Параллельная структура

В веб-приложениях ресурсы традиционно делятся по типу файлов. Разметка HTML хранится в одной папке, CSS — в другой, JavaScript — в третьей. При работе над компонентом часто приходится переключаться между этими папками. Для упрощения работы бывает удобно использовать параллельную структуру каталогов.

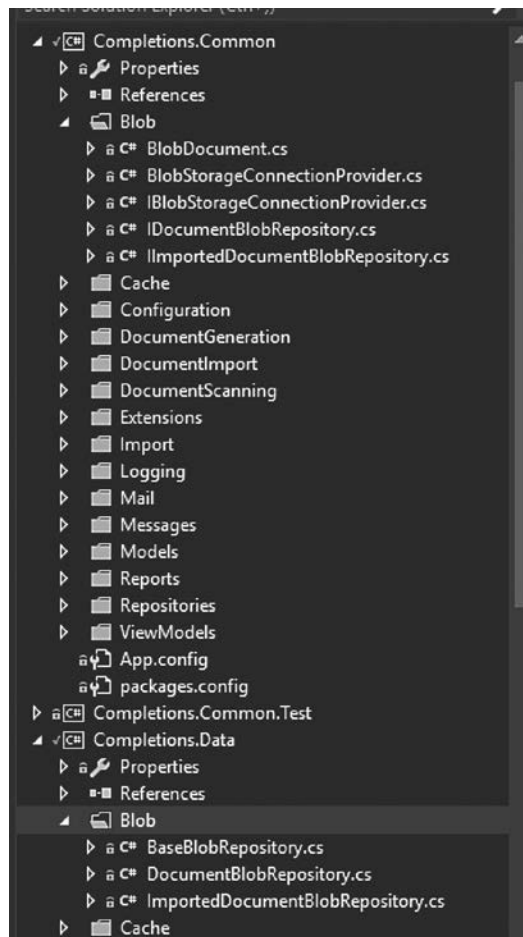


Рис. 24.1. Структура проектов в решении, включающем несколько проектов

ПРИМЕЧАНИЕ

Такие инструменты, как ReactJS, рекомендуют использовать разную организацию в зависимости от компонентов, но в большинстве проектов это перебор. Например, в проекте React код CSS, JavaScript и разметка хранятся в одном файле, что отменяет необходимость в хранении ресурсов в отдельных каталогах. Во многих отношениях такая структура более осмысленна с точки зрения повторного использования кода. Мы скорее используем полный инкапсулированный компонент, чем только CSS или JavaScript.

Этот подход распространяется на все приложение. Помимо хранения веб-ресурсов в параллельной структуре в ней также должны храниться модульные и интеграционные тесты. В частности, этот подход используется шаблоном по умолчанию: структура каталогов повторяет структуру имен контроллеров.

Паттерн «Посредник»

Существует много вариантов организации кода в проекте. Один из хорошо масштабируемых методов основан на паттерне «Посредник» (Mediator), разрывающем логические привязки между компонентами.

В паттерне «Посредник» ответственность за взаимодействие перемещается в класс-посредник. Рассмотрим пример возможной работы этого паттерна. Каждый раз, когда лыжники сканируют свои абонементы в Alpine Ski House, необходимо обратиться к базе данных и проверить, действителен ли абонемент. В то же время результат сканирования должен быть сохранен в базе данных для сбора метрик действительности абонемента. Код выглядит так:

```
[HttpPost]
public async Task<IActionResult> Post([FromBody]Scan scan)
{
    _context.Scans.Add(scan);
    await _context.SaveChangesAsync();

    if(_cardValidator.IsValid(scan.CardId, scan.LocationId))
        return StatusCode((int)HttpStatusCode.Created);
    return StatusCode((int)HttpStatusCode.Unauthorized);
}
```

Проблема в том, что этот метод пытается решать пару задач одновременно. Лучше было бы соблюдать принцип единственной обязанности и сократить метод до одной задачи; следовательно, часть функциональности следует выделить с ослаблением привязок между классами. Для этого можно создать два новых сервиса: один просто сохраняет в журнале информацию о сканировании, а другой отвечает за проверку карты на действительность. Контроллер по-прежнему отвечает за преобразование ответа от подсистемы проверки карт в код статуса HTTP. Возвращаемое значение метода контроллера зависит от значения, возвращаемого подсистемой проверки карт, но информация сканирования всегда сохраняется в базе данных. Такое описание похоже на паттерн «Передача сообщений».

Краткое введение в паттерн «Передача сообщений»

«Передача сообщений» (Messaging) — старый паттерн; некоторые специалисты считают, что именно так должно было изначально работать объектно-ориентированное программирование. «Сообщением» (message) называется набор полей, содержащих некую информацию. Если ваш метод получает несколько параметров, этот набор может рассматриваться как сообщение (в данном случае `id`, `date` и `numberOfSkiers`).

```
public ValidationResult(int id, DateTime date, int numberOfSkiers)
```

Создание объекта сообщения во всех без исключения методах приложения сопряжено со значительными затратами, поэтому мы допускаем эту деструктурированную форму как сокращенную запись. В распределенной системе сообщения часто определяются более формально и непосредственно передаются по каналу связи между двумя сервисами (иногда даже с использованием шины сообщений, обеспечивающей надежный транспортный механизм).

Сообщения делятся на две категории: команды и события. *Команда* приказывает что-то сделать, а ее имя обычно записывается в повелительной форме: `LogSkiCardSwipe`, `DeleteUser`, `AddLocation` и т. д. Недопустимая команда, как и любой другой недопустимый приказ, может не выполняться (наверное, вам не стоит показывать эту часть книги своему начальству). С другой стороны, *события* уведомляют о чем-то, что состоялось в прошлом: `SkiCardSwipeLogged`, `UserDeleted` и `LocationAdded` — эквивалентные события для команд, упоминавшихся выше. Команда обрабатывается некоторым сервисом, который обычно называется *обработчиком команд*. После того как команда была успешно выполнена, обработчик может опубликовать уведомление или событие, указывающее на факт выполнения действия.

На событие могут подписаться любое количество других сервисов. Событие отправляется из одного места после обработки команды, а его получатели находятся во многих местах. С другой стороны, команда может отправляться из нескольких мест (например, пользователь может быть удален администратором, а может удалить сам себя), но обрабатывается только одним обработчиком команды.

Реализация паттерна «Посредник»

Есть несколько вариантов реализации паттерна «Посредник». Самый простой и наименее надежный — перечисление всех компонентов, которые могут быть заинтересованы в обработке некоторого типа сообщения, в одном огромном классе-посреднике. Класс-посредник даже можно разбить по типу сообщений. Тем не менее принципиально при этом ничего не изменяется (рис. 24.2).

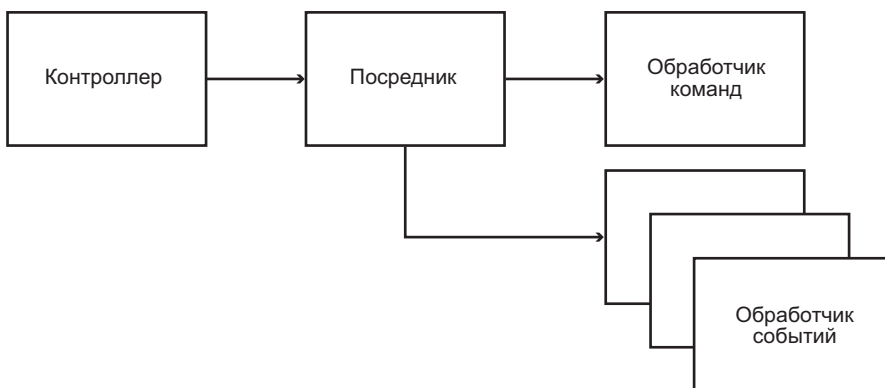


Рис. 24.2. Простая реализация паттерна «Посредник»

Остается класс, которому известны обработчики команд и событий, а именно этого хотелось бы избежать. Класс не должен знать о конкретных реализациях; и, казалось бы, эта проблема была в основном решена в главе 14 «Внедрение зависимостей». Нам нужно другое: механизм регистрации обработчиков команд и событий в контейнере и их автоматического включения в систему обработки.

К счастью, существует превосходная реализация паттерна «Посредник», которая называется MediatR. Она берет на себя все подробности связывания обработчиков с событиями. Ее интеграция в приложение Alpine Ski House начинается с регистрации фреймворка внедрения зависимостей. Нужный вам пакет NuGet называется MediatR. Затем в конфигурацию контейнера добавляется следующий код:

```
services.AddScoped<SingleInstanceFactory>(p => t => p.GetRequiredService(t));
services.AddScoped<MultiInstanceFactory>(p => t => p.GetServices(t));
services.AddMediatR(typeof(Startup).GetTypeInfo().Assembly);
```

Код регистрирует посредника, а также две фабрики, создающие обработчики команд и событий. Затем необходимо создать отправляемую команду. Создайте в корневом каталоге проекта новый каталог с именем **Features**. В нем нужно задать разумную структуру каталогов. Вопрос о том, что считать разумным, зависит от конкретного проекта. Возможно, обработчики нужно разделить по ограниченным контекстам или по сущностям. На рис. 24.3 показана структура каталога **Features** в приложении Alpine Ski House. Проект уже применяет ограниченные контексты при изоляции контекстов данных, и мы воспользуемся этим способом логической группировки для упорядочения функциональности.

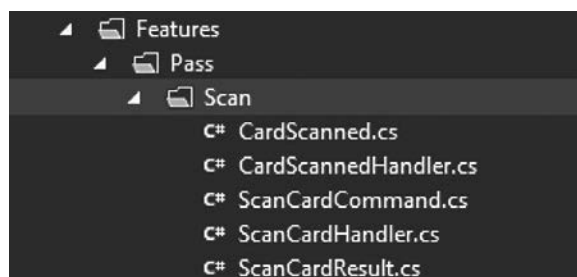


Рис. 24.3. Структура каталога Features

Посмотрим, какие классы участвуют в реализации паттерна. Первый — класс `ScanCardCommand`.

```
public class ScanCardCommand : IAsyncRequest<ScanCardResult>
{
    public int CardId { get; set; }
    public int LocationId { get; set; }
    public DateTime DateTime { get; set; }
}
```

Команда содержит все поля, которые заполняются на стороне клиента. Команды реализуют интерфейс `IAsyncRequest` и получают обобщенный параметр с результатом команды. В данном случае это `ScanCardResult` — очень простой объект передачи данных (DTO, Data Transfer Object).

```
public class ScanCardResult
{
    public bool CardIsValid { get; set; }
}
```

Пожалуй, самой интересной частью этого процесса является обработчик команд.

```
public class ScanCardHandler : IAsyncRequestHandler<ScanCardCommand,
ScanCardResult>
{
    private readonly ICardValidator _validator;
    private readonly IMediator _mediator;
    public ScanCardHandler(ICardValidator validator, IMediator mediator)
    {
        _mediator = mediator;
        _validator = validator;
    }

    public async Task<ScanCardResult> Handle(ScanCardCommand message)
    {
        await _mediator.PublishAsync(new CardScanned { CardId = message.CardId,
LocationId = message.LocationId, DateTime = message.DateTime });
        return await Task.FromResult(new ScanCardResult { CardIsValid = _validator.
IsValid(message.CardId, message.LocationId) });
    }
}
```

Конструктор использует свое участие во внедрении зависимостей для извлечения экземпляра `ICardValidator` и самого посредника. В идеале приложение должно быть асинхронным от начала до конца, поэтому используются асинхронные интерфейсы MediatR. В асинхронном методе `Handle` публикуется событие `CardScanned` (которое MediatR называет «уведомлением»), после чего происходит проверка, а результат `ScanCardResult` возвращается вызывающей стороне.

Посредник отвечает за передачу публикуемого события всем заинтересованным сторонам. В нашем примере заинтересованная сторона только одна — `CardScannedHandler`:

```
public class CardScannedHandler : IAsyncNotificationHandler<CardScanned>
{
    private readonly PassContext _context;
    public CardScannedHandler(PassContext context)
    {
        _context = context;
    }
}
```

```
public async Task Handle(CardScanned notification)
{
    var scan = new Models.Scan
    {
        CardId = notification.CardId,
        DateTime = notification.DateTime,
        LocationId = notification.LocationId
    };
    _context.Scans.Add(scan);
    await _context.SaveChangesAsync();
}
```

Класс реализует интерфейс `IAsyncNotificationHandler` для события `CardScanned`. По этому факту MediatR узнает о необходимости его подключения. В методе `Handle` оповещение используется для сохранения информации в базе данных. Вы можете создать несколько обработчиков, если нужно выполнять несколько операций, и каждый обработчик должен иметь очень узкую и легко тестируемую специализацию.

В этой реализации есть несколько возможностей для усовершенствования. Первая — имя `CardScannedHandler`. Имя ничего не сообщает о том, что делает обработчик, и если в программе есть несколько обработчиков, заинтересованных в одном событии, по такому имени его будет трудно отличить от других. Лучше было бы назвать его `LogCardScanToDatabase`. Порядок кода в обработчике команд тоже выглядит сомнительно. Должно ли событие `CardScannedEvent` инициироваться до того, как карта будет проверена? Возможно. А может быть, вы пропустили другое событие (например, `ValidCardScannedEvent` или `InvalidCardScannedEvent`). MediatR учитывает отношения полиморфизма, поэтому если `ValidCardScannedEvent` и `InvalidCardScannedEvent` расширяют `CardScannedEvent`, при генерации производного события также будут активизированы все подписчики базового события.

Возможно, вас интересует, что произойдет с контроллером теперь, когда вся логика выделена в пару обработчиков? Похвальная любознательность. Метод действия в контроллере существенно уменьшился, и теперь он выглядит так:

```
[HttpPost]
public async Task<IActionResult> Post([FromBody]ScanCardCommand scan)
{
    if ((await _mediator.SendAsync(scan)).CardIsValid)
        return StatusCode((int)HttpStatusCode.Created);
    return StatusCode((int)HttpStatusCode.Unauthorized);
}
```

Как видите, метод стал предельно компактным. Для него элементарно пишутся тесты, и он имеет всего одну обязанность — преобразование `ScanCardResult` в код статуса HTTP.

Использование паттерна «Посредник» способствует разделению обязанностей и позволяет группировать действия. Возможно, кто-то захочет пойти еще дальше и разместить контроллеры и представления, относящиеся к некоторому функциональному аспекту, в одном каталоге с событиями и командами. Так как это место для хранения не является стандартным, вероятно, чтобы эта схема работала, вам придется переопределить некоторые соглашения ASP.NET Core MVC.

ПРИМЕЧАНИЕ

Работа фреймворка ASP.NET MVC основана на соглашениях, и если вы следуете этим соглашениям, все должно складываться хорошо. Тем не менее не все проекты должны следовать одним и тем же соглашениям. В таких ситуациях вам может пригодиться интерфейс `IApplicationModelConvention`. У группы ASP.NET Monsters есть видеоролик на эту тему по адресу <https://channel9.msdn.com/Series/aspnetmonsters/ASPNET-Monsters-Ep-68-Creating-Custom-Conventions-for-ASPNET-Core-MVC>.

Области

Очень большие приложения иногда приходится делить на меньшие части, чтобы создать хотя бы видимость порядка. Области (areas), встроенные в ASP.NET MVC, предоставляют возможность вложения организационной структуры по типу матрешек: в корне иерархии проекта располагается папка **Areas**, которая содержит одну или несколько разных областей. В каждой области присутствуют папки **Controllers**, **Views** и **Models**, как и на корневом уровне (рис. 24.4).

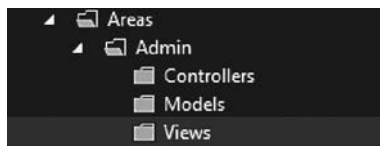


Рис. 24.4. Структура папок внутри области

Благодаря волшебству маршрутизации вы сможете обращаться к этим контроллерам после того, как они будут помечены атрибутом **Area**. Например, класс `UserController` в области **Admin** выглядит так:

```
[Area("Admin")]  
public class UserController
```

Также обновите таблицу маршрутизации, прилагаемую к шаблону по умолчанию, и включите в нее маршрут для новых областей.

```
app.UseMvc(routes =>  
{  
    routes.MapRoute(  
        "Default", "{controller}/{action}/{id}",  
        new { controller = "Home", action = "Index", id = UrlParameter.Optional }  
    );  
});
```

```
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}";
    routes.MapAreaRoute("defaultArea", "Admin", "{area}/{controller}/
        {action=Index}/{id?}");
});
```

После определения нового маршрута вы сможете осуществлять маршрутизацию к контроллерам внутри областей.

Как эта схема сочетается с идеей каталогов **Features**? В больших проектах они могут использоваться одновременно. Впрочем, необходимость в их одновременном использовании должна органично проявляться в ходе сборки приложения. Как правило, решение об их одновременном использовании не должно приниматься с первого дня.

ПРИМЕЧАНИЕ

Стив Смит (Steve Smith) написал для MSDN Magazine замечательную статью по большей части материала, представленного в этой главе. Если вас интересует эта тема и вы захотите увидеть больше примеров кода, начните с его статьи по адресу <https://msdn.microsoft.com/magazine/mt763233>.

Итоги

Организация и структура кода — сложная тема. Вполне вероятно, что методы, представленные в этой главе, не подойдут идеально для вашей конкретной ситуации. Важно постоянно размышлять над тем, какие «болевы́е точки» встречаются в вашем приложении и как улучшить структуру кода для решения этих проблем. Хотя современные интегрированные среды разработки и даже хорошие текстовые редакторы предоставляют замечательные инструменты поиска и навигации в коде, проще и естественнее ориентироваться в нем самостоятельно. Работая над кодом, помните: следующим человеком, которому придется разбираться в вашем коде через полгода, можете оказаться вы сами.

Послесловие

В комнате чувствовалась атмосфера усталой эйфории. Даниэль не ощущала ничего подобного уже много лет — пожалуй, с тех времен, когда она занималась лыжным спортом. На последней неделе команда справилась со всеми препятствиями, и приложение выглядело великолепно. Оно делало все, что положено, и делало с блеском. Даниэль не могла бы гордиться сделанным за 4 недели в большей степени... разве что если бы ее ребенком уронили в котел с эликсиром гордости.

Команда уже сидела по одну сторону стола, когда зашла руководящая группа. Измученные разработчики разительно отличались от загорелых любителей гольфа: начальство выглядело довольным и беззаботным, а пара разработчиков были небритыми. Даниэль знала, что компании не нравятся проекты, из-за которых люди допоздна задерживались на работе, а Тиму пришлось приложить немалые усилия, чтобы убедить руководство: сегодняшний «аврал» должен был стать последним.

«Если 40 часов в неделю было достаточно для Генри Форда, то и для Alpine Ski House этого должно хватить», — сказал он.

Балаш вышел с компьютером, несколькими RFID-датчиками и сканером. Он подождал, пока гольфисты притихнут и начнут слушать.

«Месяц назад, — начал он, — мы взялись за невозможный проект: проектирование целой системы и запуск ее в эксплуатацию всего за четыре недели. Откровенно говоря, в начале проекта мы с Тимом основательно прорабатывали стратегии управления ожиданиями и объяснения того, почему мы не уложились в срок. Это было до того, как я познакомился с командой и понял, на что она способна».

«Эти люди хорошо сработались. Используя совершенно новый стек технологий, они создали решение всех изначально поставленных задач. Время, потраченное на управление ожиданиями, было потрачено даром».

После этого Балаш начал объяснять, как пользователи будут вводить свои данные в приложении и как они будут покупать абонементы разных типов. Похоже, на гольфистов это произвело впечатление; они задали пару вопросов, но большую часть времени сидели молча. Возможно, им просто хотелось сбежать с собрания и вернуться на поле для гольфа.

С утра было морозно, а лыжный сезон начался всего месяц назад. В отделе кадров уже лежали кипы резюме на должности операторов подъемников или лыжных инструкторов.

Даниэль снова стала прислушиваться, когда Балаш заговорил о локализации. «Да, Фрэнк, — говорил он, — мы об этом подумали. Я понимаю, что сюда приходит много людей, говорящих только на французском. Поддержки других языков пока нет, но мы собираемся ею заняться. Видишь ли, Фрэнк, такие проекты никогда не заканчиваются, потому что всегда есть место для улучшений. Я общался с бизнес-стороной, в том числе и с многими из вас, — Балаш указал на гольфистов, — чтобы узнать, какие новые возможности нужно включить в приложение. Я составил довольно подробный список, и мне еще придется потратить время на расстановку приоритетов. Я бы сказал, что здесь еще работы на несколько месяцев».

«И это, — подключился Тим, не склонный к молчанию, — только начало. У нас есть несколько других приложений, которые мы собирались заменить коммерческими продуктами. Вот только с коммерческими продуктами мы в лучшем случае выйдем на уровень конкурентов, а собственная группа разработки, которая пишет программы под заказ, даст нам конкурентное преимущество. В наше время нетехнологических компаний просто нет. Либо внедряй инновации, либо погибай». Казалось, Тим собирается говорить на эту тему довольно долго, но его прервал один из гольфистов. Даниэль узнала в нем председателя совета директоров.

«Спасибо, Тим, при том что мы видели сегодня, я с тобой, пожалуй, соглашусь. — Окружающие согласно покивали. — Проект Parsley — именно то, что нам было нужно, и в перспективе я вижу увеличение доли внутренних разработок. Пожалуйста, продолжай, Балаш».

Разработчики согласно кивнули. Последние несколько недель Даниэль особенно не задумывалась о поиске новой работы — она была слишком сильно занята текущими делами.

Балаш продолжал свою демонстрацию. Всем особенно понравились сканеры RFID. «И что, люди просто идут на подъемник?» — спросил кто-то из гольфистов.

«Угу, — ответил Балаш. — Система довольно надежная. Мы экономим время на проверке абонементов, это экономит нам деньги, а лыжники намного быстрее попадают на склон».

Наконец, демонстрация закончилась, и гольфисты сразу заговорили о возможностях. «Систему можно использовать для покупки подарочных карт? Сможем ли мы принимать платежи в кафетериях по тем же картам? А если светодиодное табло будет приветствовать каждого посетителя по имени?»

Это был хороший знак. Гольфисты уже согласились с тем, что разработка приложения прошла успешно, и уже думали, что в него можно добавить. Балаш отвечал на многочисленные вопросы и вскоре начал приглашать всех спрашивающих зайти к нему на следующей неделе. Было совершенно непонятно, над чем же команда будет работать дальше. Цена успеха, подумала Даниэль. В итоге гольфисты поняли, что им не удастся уговорить Балаша согласиться на вклю-

чение новой функциональности «на месте», и покинули собрание на полчаса позже запланированного.

Балаш повернулся к команде и сказал: «Думаю, все прошло неплохо. Все идет к тому, что мы сможем продолжить работу над проектом. Я никогда в вас не сомневался».

«Никогда? — спросил Марк-2. — Разве ты не сказал им, что сомневался в нас поначалу?»

«Ммм, почти никогда не сомневался, — вывернулся Балаш. — Когда не знал никого из вас».

«Ладно, ладно. — сказал Тим, повторяя любимую фразу Балаша. — Думаю, наша работа закончена. А теперь все свободны. Я не хочу здесь видеть никого из вас до понедельника».

«Хорошо, не забудьте оставить утро понедельника свободным, чтобы мы могли обсудить вторую фазу проекта. Я назову ее "проект «Горчица»", — сказал Балаш, когда все уже направлялись к выходу, — потому что это будет жаркая штука. Народ? Народ?»

Но в комнате уже никого не было.

Об авторах



Джеймс Чамберс, пятикратный обладатель статуса Microsoft MVP в области технологий разработки, в настоящее время занимается разработкой с использованием ASP.NET Core и MVC Framework на платформе Azure и AWS. Он является независимым консультантом, преподавателем и активным блоггером, а также участником ряда проектов с открытым кодом.



Дэвид Пэккетт, четырехкратный обладатель статуса Microsoft MVP — разработчик ПО и независимый консультант. Он обладает обширным опытом использования .NET для построения приложений Windows и веб-приложений, глубокими техническими познаниями и страстью к качественным пользовательским интерфейсам.



Саймон Тиммс, обладатель статуса Microsoft MVP на протяжении многих лет — организатор сообществ, блоггер, разработчик и независимый консультант. Его технологические интересы весьма разнообразны, он увлекается всем — от распределенных систем до новомодных фреймворков JavaScript. Обладая хорошей подготовкой как в области разработки, так и в организации текущей работы, он нередко сводит с ума свои группы, участвуя во всех этапах, от построения и разработки до обслуживания серверов.