

Содержание

Об авторе	22
Часть I. Введение в ASP.NET Core	23
Глава 1. Основы ASP.NET Core	24
Понятие ASP.NET Core	24
Прикладные инфраструктуры	24
Служебные инфраструктуры	26
Платформа ASP.NET Core	27
Что необходимо для чтения этой книги?	28
Какое программное обеспечение необходимо для проработки примеров?	28
Какая платформа необходима для проработки примеров?	28
Что делать, если при проработке примеров возникают проблемы?	28
Что эта книга охватывает?	28
Что эта книга не охватывает?	29
Резюме	29
Глава 2. Начало работы	30
Выбор редактора кода	30
Установка Visual Studio	30
Установка Visual Studio Code	33
Установка SQL Server LocalDB	34
Создание проекта ASP.NET Core	37
Открытие проекта с использованием Visual Studio	38
Открытие проекта с использованием Visual Studio Code	39
Запуск приложения ASP.NET Core	40
Понятие конечных точек	41
Понятие маршрутов	43
Понятие визуализации HTML-разметки	44
Собираем фрагменты вместе	48
Резюме	48
Глава 3. Ваше первое приложение ASP.NET Core	49
Предварительная настройка	49
Создание проекта	49
Добавление модели данных	51
Создание второго действия и представления	52
Связывание с методами действий	53
Построение формы	54
Получение данных формы	56
Добавление представления с благодарностью	59
Отображение ответов	60
Добавление проверки достоверности	62
Стилизация содержимого	67
Резюме	73
Глава 4. Использование инструментов для разработки	74
Создание проектов ASP.NET Core	75
Создание проекта с использованием командной строки	75

Создание проекта с использованием Visual Studio	79
Добавление кода и содержимого в проект	81
Понятие генерации шаблонов элементов	82
Компиляция и запуск проектов	83
Компиляция и запуск проектов с использованием командной строки	84
Компиляция и запуск проектов с использованием Visual Studio Code	84
Компиляция и запуск проектов с использованием Visual Studio	85
Управление пакетами	86
Управление пакетами NuGet	86
Управление пакетами инструментов	87
Управление пакетами стороны клиента	87
Управление пакетами с использованием Visual Studio	89
Отладка проектов	90
Резюме	91
Глава 5. Важные функциональные возможности языка C#	92
Подготовка проекта для примера	93
Открытие проекта	93
Включение инфраструктуры MVC	93
Создание компонентов приложения	94
Выбор порта HTTP	96
Запуск примера приложения	96
Использование null-условной операции	97
Связывание в цепочки null-условных операций	98
Комбинирование null-условной операции и операции объединения с null	99
Использование автоматически реализуемых свойств	101
Использование инициализаторов автоматически реализуемых свойств	102
Создание автоматически реализуемых свойств, предназначенных только для чтения	103
Использование интерполяции строк	104
Использование инициализаторов объектов и коллекций	105
Использование инициализатора индексированной коллекции	106
Сопоставление с образцом	108
Сопоставление с образцом в операторах switch	109
Использование расширяющих методов	110
Применение расширяющих методов к интерфейсу	111
Создание фильтрующих расширяющих методов	113
Использование лямбда-выражений	114
Определение функций	116
Использование методов и свойств в форме лямбда-выражений	119
Использование автоматического вывода типа и анонимных типов	121
Использование анонимных типов	121
Использование стандартных реализаций в интерфейсах	123
Использование асинхронных методов	126
Работа с задачами напрямую	126
Применение ключевых слов async и await	127
Использование асинхронного перечисления	129
Получение имен	132
Резюме	133

Глава 6. Тестирование приложений ASP.NET Core	134
Подготовка проекта для примера	135
Открытие проекта	135
Выбор порта HTTP	136
Включение инфраструктуры MVC	136
Создание компонентов приложения	137
Запуск примера приложения	138
Создание проекта модульного тестирования	139
Удаление стандартного тестового класса	140
Написание и прогон модульных тестов	140
Прогон тестов с помощью проводника тестов Visual Studio	142
Прогон тестов с помощью Visual Studio Code	143
Прогон тестов с помощью командной строки	143
Исправление модульного теста	144
Изолирование компонентов для модульного тестирования	145
Использование инфраструктуры имитации	149
Создание имитированного объекта	150
Резюме	151
Глава 7. SportsStore: реальное приложение	152
Создание проектов	153
Создание проекта модульного тестирования	153
Создание папок проекта приложения	154
Открытие проектов	154
Подготовка служб приложения и конвейера запросов	155
Конфигурирование механизма визуализации Razor	157
Создание контроллера и представления	158
Начало создания модели данных	159
Проверка и запуск приложения	159
Добавление данных в приложение	160
Установка пакетов Entity Framework Core	160
Определение строки подключения	161
Создание класса контекста базы данных	162
Конфигурирование Entity Framework Core	162
Создание хранилища	163
Создание миграции базы данных	166
Создание начальных данных	167
Отображение списка товаров	170
Подготовка контроллера	170
Обновление представления	172
Запуск приложения	173
Добавление разбиения на страницы	173
Отображение ссылок на страницы	175
Улучшение URL	183
Стилизация содержимого	185
Установка пакета Bootstrap	185
Применение стилей Bootstrap	186
Создание частичного представления	189
Резюме	190

Глава 8. SportsStore: навигация и корзина	191
Добавление навигационных элементов управления	191
Фильтрация списка товаров	191
Улучшение схемы URL	196
Построение меню навигации по категориям	200
Корректировка счетчика страниц	207
Построение корзины для покупок	210
Конфигурирование Razor Pages	210
Создание страницы Razor	212
Создание кнопок добавления в корзину	213
Включение поддержки сеансов	215
Реализация средства корзины	216
Завершение страницы Razor	220
Резюме	225
Глава 9. SportsStore: завершение корзины	226
Усовершенствование модели корзины с помощью службы	226
Создание класса корзины, осведомленного о хранилище	226
Регистрация службы	228
Упрощение Razor-страницы Cart	230
Завершение функциональности корзины	232
Удаление элементов из корзины	232
Добавление виджета с итоговой информацией по корзине	234
Отправка заказов	237
Создание класса модели	237
Добавление реализации процесса оплаты	238
Создание контроллера и представления	239
Реализация обработки заказов	241
Завершение построения контроллера Order	244
Отображение сообщений об ошибках проверки достоверности	248
Отображение итоговой страницы	249
Резюме	250
Глава 10. SportsStore: администрирование	251
Подготовительная работа для включения Blazor Server	251
Создание файла импортирования	253
Создание начальной страницы Razor	253
Создание компонентов маршрутизации и компоновки	254
Создание компонентов Razor	255
Проверка корректности настройки Blazor	256
Управление заказами	256
Расширение модели	257
Отображение заказов для администратора	258
Добавление управления каталогом	261
Расширение хранилища	262
Применение атрибутов проверки достоверности в модели данных	263
Создание компонента для отображения списка	263
Создание компонента для отображения деталей	265
Создание компонента для редактирования	266
Удаление товаров	268
Резюме	270

Глава 11. SportsStore: защита и развертывание	271
Защита средств администрирования	271
Создание базы данных Identity	271
Установка пакета с поддержкой ASP.NET Core Identity для Entity Framework Core	272
Добавление традиционного средства администрирования	277
Применение базовой политики авторизации	278
Создание контроллера и представлений для учетных записей	280
Тестирование политики безопасности	283
Подготовка к развертыванию приложения ASP.NET Core	284
Конфигурирование обработки ошибок	284
Создание настроек конфигурации для производственной среды	286
Создание образа Docker	286
Запуск контейнерного приложения	289
Резюме	290
Часть II. Платформа ASP.NET Core	291
Глава 12. Введение в платформу ASP.NET Core	292
Подготовительная работа	293
Запуск примера приложения	294
Освоение платформы ASP.NET Core	295
Понятие промежуточного программного обеспечения и конвейера запросов	295
Понятие служб	295
Освоение проекта ASP.NET Core	296
Понятие точки входа	298
Понятие класса Startup	299
Понятие файла проекта	300
Создание специального промежуточного программного обеспечения	302
Определение промежуточного программного обеспечения с использованием класса	306
Понятие обратного пути конвейера	308
Короткое замыкание конвейера запросов	310
Создание ветвей конвейера	311
Создание терминального промежуточного программного обеспечения	314
Конфигурирование промежуточного ПО	316
Использование шаблона параметров с промежуточным программным обеспечением на основе классов	318
Резюме	320
Глава 13. Использование маршрутизации URL	321
Подготовительная работа	322
Понятие маршрутизации URL	325
Добавление промежуточного программного обеспечения маршрутизации и определение конечной точки	326
Понятие шаблонов URL	329
Использование переменных сегментов в шаблонах	330
Генерирование URL из маршрутов	335
Управление сопоставлением URL	339
Сопоставление множества значений из одиночного сегмента URL	339
Использование стандартных значений для переменных сегментов	340

Использование необязательных сегментов в шаблоне URL	341
Использование переменной сегмента общего захвата	343
Ограничение сопоставления сегментов	344
Ограничение сопоставления специфическим набором значений	347
Определение запасных маршрутов	347
Расширенные возможности маршрутизации	349
Создание специальных ограничений	349
Избегание исключений, связанных с неоднозначными маршрутами	351
Доступ к конечной точке в компоненте промежуточного программного обеспечения	353
Резюме	355
Глава 14. Использование внедрения зависимостей	356
Подготовительная работа	357
Создание компонента промежуточного программного обеспечения и конечной точки	358
Конфигурирование конвейера запросов	359
Понятие местоположения служб и сильной связи	360
Проблема местоположения служб	361
Проблема сильно связанных компонентов	363
Использование внедрения зависимостей	366
Использование службы в классе промежуточного программного обеспечения	367
Использование службы в конечной точке	369
Получение служб из объекта <code>HttpContext</code>	369
Использование функции адаптера	370
Использование служебного класса активации	371
Использование жизненных циклов служб	374
Создание служб с переходным жизненным циклом	375
Избегание попадания в ловушку с многократным использованием служб с переходным жизненным циклом	376
Использование служб с жизненным циклом, ограниченным областью действия	379
Создание новых обработчиков для каждого запроса	383
Использование служб с жизненным циклом, ограниченным областью действия, в лямбда-функциях	384
Другие возможности внедрения зависимостей	385
Создание цепочек зависимостей	385
Доступ к службам в методе <code>ConfigureServices</code>	387
Использование фабричных функций служб	388
Создание служб с множеством реализаций	390
Использование несвязанных типов в службах	393
Резюме	394
Глава 15. Использование функциональных средств платформы, часть 1	395
Подготовительная работа	396
Использование службы конфигурации	397
Конфигурационный файл, специфичный для среды	398
Доступ к настройкам конфигурации	399
Использование данных конфигурации в службах	401
Файл настроек запуска	404
Выяснение среды в классе <code>Startup</code>	409

12 Содержание

Хранение секретов пользователя	410
Сохранение секретов пользователя	411
Чтение секретов пользователя	412
Использование службы ведения журнала	414
Генерирование журнальных сообщений	414
Конфигурирование минимальных уровней ведения журнала	418
Использование статического содержимого и пакетов стороны клиента	419
Добавление промежуточного программного обеспечения для статического содержимого	420
Изменение стандартных параметров промежуточного программного обеспечения для статического содержимого	421
Использование пакетов стороны клиента	424
Использование пакета стороны клиента	426
Резюме	427
Глава 16. Использование функциональных средств платформы, часть 2	428
Подготовительная работа	428
Использование cookie-наборов	430
Включение проверки согласия на отслеживание cookie-наборов	432
Координация согласия на отслеживание cookie-наборов	434
Использование сеансов	437
Конфигурирование служб и промежуточного программного обеспечения сеансов	437
Использование данных сеанса	440
Работа с подключениями HTTPS	442
Разрешение подключений HTTPS	442
Обнаружение запросов HTTPS	444
Принудительное применение запросов HTTPS	445
Включение протокола HSTS	447
Обработка исключений и ошибок	449
Возвращение HTML-ответа с сообщением об ошибке	451
Улучшение ответов с кодом состояния	454
Фильтрация запросов с использованием заголовка Host	456
Резюме	458
Глава 17. Работа с данными	459
Подготовительная работа	460
Кеширование данных	463
Кеширование значений данных	464
Использование разделяемого и постоянного кеша данных	468
Кеширование ответов	472
Использование Entity Framework Core	475
Установка Entity Framework Core	476
Создание модели данных	477
Конфигурирование службы базы данных	478
Создание и применение миграции базы данных	480
Создание начальных данных	480
Использование данных в конечной точке	484
Резюме	486

Часть III. Приложения ASP.NET Core	487
Глава 18. Создание проекта для примера приложения	488
Создание проекта	488
Добавление модели данных	489
Добавление пакетов NuGet в проект	489
Создание модели данных	489
Подготовка начальных данных	491
Конфигурирование служб и промежуточного программного обеспечения Entity Framework Core	492
Создание и применение миграции	494
Добавление инфраструктуры CSS	494
Конфигурирование конвейера запросов	494
Запуск примера приложения	496
Резюме	496
Глава 19. Создание веб-служб REST	497
Подготовительная работа	498
Понятие веб-служб REST	499
Понятие URL и методов запросов	499
Понятие JSON	500
Создание веб-службы с использованием специальной конечной точки	500
Создание веб-службы с использованием контроллера	503
Включение инфраструктуры MVC Framework	504
Создание контроллера	505
Понятие базового класса	506
Понятие атрибутов контроллера	507
Совершенствование веб-службы	515
Использование асинхронных действий	516
Предотвращение чрезмерной привязки	518
Использование результатов действий	519
Проверка достоверности данных	526
Применение атрибута ApiController	527
Пропуск свойств со значениями null	529
Резюме	531
Глава 20. Расширенные средства для веб-служб	532
Подготовительная работа	533
Удаление базы данных	534
Запуск примера приложения	534
Работа со связанными данными	534
Разрыв циклических ссылок в связанных данных	536
Поддержка HTTP-метода PATCH	537
Понятие стандарта JSON Patch	538
Установка и конфигурирование пакета JSON Patch	538
Определение метода действия	539
Понятие форматирования содержимого	541
Понятие стандартной политики содержимого	541
Понятие согласования содержимого	543

14 Содержание

Указание формата результата действия	547
Запрашивание формата в URL	548
Ограничение форматов, получаемых методом действия	549
Документирование и выявление веб-служб	551
Разрешение конфликтов между действиями	551
Установка и конфигурирование пакета Swashbuckle	552
Точная настройка описания OpenAPI	555
Резюме	558
Глава 21. Использование контроллеров с представлениями, часть 1	559
Подготовительная работа	560
Удаление базы данных	561
Запуск примера приложения	561
Начало работы с представлениями	562
Конфигурирование приложения	562
Создание контроллера HTML	563
Создание представления Razor	567
Выбор представления по имени	569
Работа с представлениями Razor	573
Установка типа модели представления	576
Синтаксис Razor	581
Понятие директив	581
Понятие выражений содержимого	582
Установка содержимого элементов	583
Установка значений атрибутов	584
Использование условных выражений	585
Перечисление последовательностей	589
Использование кодовых блоков Razor	591
Резюме	592
Глава 22. Использование контроллеров с представлениями, часть 2	593
Подготовительная работа	594
Удаление базы данных	595
Запуск примера приложения	595
Использование объекта ViewBag	596
Использование средства временных данных	598
Работа с компоновками	600
Конфигурирование компоновок с использованием объекта ViewBag	603
Использование файла начала представлений	604
Переопределение стандартной компоновки	606
Использование разделов компоновки	610
Использование частичных представлений	617
Включение частичных представлений	617
Создание частичного представления	617
Применение частичного представления	618
Кодирование содержимого	621
Кодирование HTML	621
Кодирование JSON	623
Резюме	624

Глава 23. Использование инфраструктуры Razor Pages	625
Подготовительная работа	626
Запуск примера приложения	627
Понятие инфраструктуры Razor Pages	627
Конфигурирование Razor Pages	628
Создание страницы Razor	629
Маршрутизация в инфраструктуре Razor Pages	633
Указание шаблона маршрутизации в странице Razor	635
Добавление маршрутов для страницы Razor	637
Понятие класса модели страницы	639
Использование файла отделенного кода	640
Понятие результатов акций в страницах Razor	642
Обработка множества HTTP-методов	646
Выбор метода обработчика	648
Понятие представления страницы Razor	650
Создание компоновки для страниц Razor	650
Использование частичных представлений в страницах Razor	652
Создание страниц Razor без моделей страниц	654
Резюме	655
Глава 24. Использование компонентов представлений	656
Подготовительная работа	657
Удаление базы данных	659
Запуск примера приложения	660
Понятие компонентов представлений	660
Создание и использование компонента представления	661
Применение компонента представления	662
Понятие результатов компонентов представлений	665
Возвращение частичного представления	666
Возвращение фрагментов HTML-разметки	669
Получение данных контекста	672
Передача контекста из родительского представления с использованием аргументов	673
Создание асинхронных компонентов представлений	676
Создание классов компонентов представлений	678
Создание гибридного класса контроллера	681
Резюме	683
Глава 25. Использование вспомогательных функций дескрипторов	684
Подготовительная работа	685
Удаление базы данных	687
Запуск примера приложения	687
Создание вспомогательной функции дескриптора	688
Определение класса вспомогательной функции дескриптора	688
Регистрация вспомогательных функций дескрипторов	691
Использование вспомогательной функции дескриптора	692
Сужение области действия для вспомогательной функции дескриптора	693
Расширение области действия для вспомогательной функции дескриптора	695
Расширенные возможности вспомогательных функций дескрипторов	697
Создание сокращенных элементов	697

Создание элементов программным образом	700
Присоединение в начале и в конце содержимого и элементов	701
Получение данных контекста представления	704
Работа с выражениями модели	706
Координация между вспомогательными функциями дескрипторов	711
Подавление выходного элемента	713
Использование компонентов вспомогательных функций дескрипторов	714
Создание компонента вспомогательной функции дескриптора	714
Расширение выбора элементов компонентами вспомогательных функций дескрипторов	717
Резюме	718
Глава 26. Использование встроенных вспомогательных функций дескрипторов	719
Подготовительная работа	720
Добавление файла изображения	722
Установка пакета стороны клиента	722
Удаление базы данных	722
Запуск примера приложения	723
Включение встроенных вспомогательных функций дескрипторов	723
Трансформирование якорных элементов	724
Использование якорных элементов для страниц Razor	726
Использование вспомогательных функций дескрипторов для файлов JavaScript и CSS	728
Управление файлами JavaScript	728
Исключение файлов	733
Работа с элементами изображений	740
Использование кеша данных	741
Установка срока действия кеша	744
Использование вспомогательной функции дескриптора для среды размещения	747
Резюме	748
Глава 27. Использование вспомогательных функций дескрипторов для форм	749
Подготовительная работа	750
Удаление базы данных	751
Запуск примера приложения	752
Понятие шаблона обработки форм	752
Создание контроллера для обработки форм	753
Создание страницы Razor для обработки форм	756
Использование вспомогательных функций дескрипторов для улучшения HTML-форм	758
Работа с элементами form	758
Трансформирование кнопок формы	760
Работа с элементами input	761
Трансформирование атрибута type элемента input	763
Форматирование значений элементов input	765
Отображение значений из связанных данных в элементах input	769
Работа с элементами label	771
Работа с элементами select и option	774
Заполнение элемента select	775
Работа с элементами textarea	777

Использование средства противодействия подделке	779
Включение средства противодействия подделке в контроллере	781
Включение средства противодействия подделке в странице Razor	782
Использование маркеров противодействия подделке с клиентами JavaScript	783
Резюме	786
Глава 28. Использование привязки моделей	787
Подготовительная работа	788
Удаление базы данных	789
Запуск примера приложения	789
Понятие привязки моделей	789
Привязка простых типов данных	791
Привязка простых типов данных в страницах Razor	792
Понятие стандартных значений привязки	794
Привязка сложных типов	797
Привязка свойства	799
Привязка вложенных сложных типов	801
Избирательная привязка свойств	805
Привязка массивов и коллекций	808
Привязка массивов	808
Привязка простых коллекций	811
Привязка словарей	813
Привязка коллекций сложных типов	814
Указание источника привязки модели	817
Выбор источника привязки для свойства	819
Использование заголовков для привязки моделей	820
Использование тел запросов как источников для привязки	821
Привязка моделей вручную	822
Резюме	824
Глава 29. Использование проверки достоверности моделей	825
Подготовительная работа	825
Удаление базы данных	827
Запуск примера приложения	828
Потребность в проверке достоверности моделей	828
Явная проверка достоверности данных в контроллере	829
Отображение пользователю сообщений об ошибках проверки достоверности	832
Отображение сообщений об ошибках проверки достоверности	834
Отображение сообщений уровня свойств	840
Отображение сообщений уровня модели	841
Явная проверка достоверности данных в странице Razor	844
Указание правил проверки достоверности с использованием метаданных	846
Создание специального атрибута проверки достоверности для свойства	851
Выполнение проверки достоверности на стороне клиента	855
Выполнение удаленной проверки достоверности	858
Выполнение удаленной проверки достоверности в страницах Razor	861
Резюме	862

Глава 30. Использование фильтров	863
Подготовительная работа	864
Разрешение подключений HTTPS	865
Удаление базы данных	866
Запуск примера приложения	867
Использование фильтров	867
Использование фильтров в страницах Razor	871
Понятие фильтров	872
Создание специальных фильтров	873
Понятие фильтров авторизации	874
Понятие фильтров ресурсов	876
Понятие фильтров действий	881
Понятие фильтров страниц	886
Понятие фильтров результатов	890
Понятие фильтров исключений	895
Управление жизненным циклом фильтров	898
Создание фабрик фильтров	900
Использование областей действия, связанных с внедрением зависимостей, для управления жизненным циклом фильтров	902
Создание глобальных фильтров	903
Понятие и изменение порядка выполнения фильтров	905
Изменение порядка выполнения фильтров	908
Резюме	909
Глава 31. Создание приложений с формами	910
Подготовительная работа	910
Удаление базы данных	913
Запуск примера приложения	913
Создание приложения MVC с формами	914
Подготовка модели представления и представления	914
Чтение данных	916
Создание данных	917
Редактирование данных	922
Удаление данных	924
Создание приложения Razor Pages с формами	925
Создание общей функциональности	928
Определение страниц для операций CRUD	930
Создание новых объектов связанных данных	933
Предоставление связанных данных в том же самом запросе	933
Прерывание для создания новых данных	936
Резюме	940
Часть IV. Расширенные средства ASP.NET Core	941
Глава 32. Создание проекта для примера приложения	942
Создание проекта	942
Добавление пакетов NuGet в проект	942
Добавление модели данных	944
Подготовка начальных данных	945

Конфигурирование служб и промежуточного программного обеспечения	
Entity Framework Core	947
Создание и применение миграции	948
Добавление инфраструктуры Bootstrap CSS	949
Конфигурирование служб и промежуточного программного обеспечения	949
Создание контроллера и представления	950
Создание страницы Razor	953
Запуск примера приложения	955
Резюме	955
Глава 33. Использование Blazor Server, часть 1	956
Подготовительная работа	957
Понятие инфраструктуры Blazor Server	958
Преимущества Blazor Server	959
Недостатки Blazor Server	959
Выбор между Blazor Server и Angular/React/Vue.js	960
Начало работы с Blazor	960
Конфигурирование ASP.NET Core для Blazor Server	960
Создание компонента Razor	963
Базовые средства компонентов Razor	968
Понятие событий Blazor и привязок данных	968
Работа с привязками данных	977
Использование файлов классов для определения компонентов	982
Использование класс отделенного кода	982
Определение класса компонента Razor	984
Резюме	985
Глава 34. Использование Blazor Server, часть 2	986
Подготовительная работа	986
Комбинирование компонентов	987
Конфигурирование компонентов с помощью атрибутов	989
Создание специальных событий и привязок	994
Отображение дочернего содержимого в компоненте	999
Создание компонентов шаблонов	1001
Использование параметров обобщенных типов в компонентах шаблонов	1003
Каскадные параметры	1009
Обработка ошибок	1012
Обработка ошибок подключения	1012
Обработка перехваченных ошибок приложения	1015
Резюме	1017
Глава 35. Расширенные средства Blazor	1018
Подготовительная работа	1019
Использование маршрутизации компонентов	1020
Подготовка страницы Razor	1021
Добавление маршрутов к компонентам	1022
Навигация между маршрутизируемыми компонентами	1025
Получение данных маршрутизации	1028
Определение общего содержимого с использованием компоновок	1029
Методы жизненного цикла компонента	1032
Использование методов жизненного цикла для асинхронных задач	1034

Управление взаимодействием с компонентами	1036
Использование ссылок на дочерние компоненты	1036
Взаимодействие с компонентами из другого кода	1039
Взаимодействие с компонентами с использованием JavaScript	1043
Резюме	1052
Глава 36. Формы Blazor и данные	1053
Подготовительная работа	1054
Удаление базы данных и запуск примера приложения	1057
Использование компонентов форм Blazor	1057
Создание специальных компонентов форм	1060
Проверка достоверности данных формы	1062
Обработка событий формы	1067
Использование Entity Framework Core вместе с Blazor	1069
Проблема области действия контекста Entity Framework Core	1069
Проблема повторяющихся запросов	1073
Выполнение операций создания, чтения, обновления и удаления	1080
Создание компонента List	1080
Создание компонента Details	1081
Создание компонента Editor	1082
Расширение средств форм Blazor	1085
Создание специального ограничения проверки достоверности	1086
Создание компонента кнопки отправки, доступной только когда данные формы допустимы	1089
Резюме	1090
Глава 37. Использование Blazor WebAssembly	1091
Подготовительная работа	1092
Удаление базы данных и запуск примера приложения	1094
Настройка Blazor WebAssembly	1095
Создание разделяемого проекта	1095
Создание проекта Blazor WebAssembly	1095
Подготовка проекта ASP.NET Core	1096
Добавление ссылок на проекты в решение	1096
Открытие проектов	1096
Завершение конфигурирования Blazor WebAssembly	1097
Тестирование заполняющих компонентов	1099
Создание компонента Blazor WebAssembly	1100
Импортирование пространства имен для модели данных	1100
Создание компонента	1101
Создание компоновки	1104
Определение стилей CSS	1105
Завершение приложения с формами Blazor WebAssembly	1106
Создание компонента Details	1106
Создание компонента Editor	1108
Резюме	1110

Глава 38. Использование ASP.NET Core Identity	1111
Подготовительная работа	1112
Подготовка проекта к использованию ASP.NET Core Identity	1113
Подготовка базы данных ASP.NET Core Identity	1114
Конфигурирование приложения	1115
Создание и применение миграции базы данных Identity	1116
Создание инструментов управления пользователями	1117
Подготовка к созданию инструментов управления пользователями	1118
Перечисление учетных записей пользователей	1119
Создание пользователей	1120
Редактирование пользователей	1128
Удаление пользователей	1131
Создание инструментов управления ролями	1132
Подготовка к созданию инструментов управления ролями	1133
Перечисление и удаление ролей	1134
Создание ролей	1135
Управление членством в ролях	1136
Резюме	1139
Глава 39. Практическое применение ASP.NET Core Identity	1140
Подготовительная работа	1140
Аутентификация пользователей	1142
Создание средства входа	1142
Инспектирование cookie-набора ASP.NET Core Identity	1145
Создание страницы выхода	1146
Тестирование средства аутентификации	1146
Включение промежуточного программного обеспечения для аутентификации ASP.NET Core Identity	1147
Авторизация доступа к конечным точкам	1150
Применение атрибута Authorization	1150
Включение промежуточного программного обеспечения для авторизации	1151
Создание конечной точки для отказа в доступе	1152
Создание начальных данных	1152
Тестирование последовательности аутентификации	1154
Авторизация доступа к приложениям Blazor	1156
Выполнение авторизации в компонентах Blazor	1157
Отображение содержимого авторизованным пользователям	1159
Аутентификация и авторизация для веб-служб	1161
Построение простого клиента JavaScript	1164
Ограничение доступа к веб-службе	1166
Использование аутентификации, основанной на cookie-наборах	1167
Использование аутентификации, основанной на маркерах носителя	1170
Создание маркеров	1171
Аутентификация с помощью маркеров	1173
Разрешение доступа с помощью маркеров	1176
Использование маркеров для запрашивания данных	1176
Резюме	1178
Предметный указатель	1179

ГЛАВА 6

Тестирование приложений ASP.NET Core

В настоящей главе будет показано, как проводить модульное тестирование приложений ASP.NET Core. Модульное тестирование — это вид тестирования, при котором индивидуальные компоненты изолируются от остальной части приложения, позволяя тщательно проверить их поведение. Инфраструктура ASP.NET Core была спроектирована так, чтобы облегчить задачу создания модульных тестов, и имеется поддержка широкого диапазона инфраструктур модульного тестирования. В главе рассматривается подготовка проекта модульного тестирования, а также описывается процесс написания и прогона тестов. В табл. 6.1 приведена сводка по главе.

Проводить ли модульное тестирование?

Наличие возможности легко выполнять модульное тестирование является одним из преимуществ использования инфраструктуры ASP.NET Core, но такая процедура предназначена не для всех, и я не намерен притворяться, что дело обстоит иначе.

Мне нравится модульное тестирование, и я применяю его в своих проектах, но далеко не во всех и не настолько согласованно, как вы могли ожидать. Я предпочитаю концентрироваться на написании модульных тестов для средств и функций, о которых знаю, что они будут трудны в реализации и вполне вероятно станут источником ошибок после развертывания. В таких ситуациях модульное тестирование помогает структурировать мои мысли о том, как лучше всего реализовать то, что необходимо. Я считаю, что одно лишь размышление о том, что нужно тестировать, способствует появлению идей относительно потенциальных проблем, причем до того, как доведется иметь дело с действительными ошибками и дефектами.

Тем не менее, модульное тестирование — инструмент, а не догма, и только лично вы знаете, в каком объеме оно должно проводиться. Если вы не находите модульное тестирование полезным или располагаете другой методологией, которая вам больше подходит, то не должны думать, что обязаны использовать модульное тестирование просто потому, что это модно. (Однако если вы *не* располагаете более эффективной методологией и не тестируете вообще, то вероятно даете возможность пользователям обнаруживать имеющиеся ошибки, что редко оказывается идеальным решением. Вы *не* обязаны выполнять модульное тестирование, но на самом деле должны проводить хотя бы *какое-то* тестирование.)

Если вы не сталкивались с модульным тестированием ранее, тогда я предлагаю опробовать его и посмотреть, как оно работает. Если вы не поклонник модульного тестирования, то можете пропустить данную главу и перейти к чтению главы 7, где начнется построение более реалистичного приложения ASP.NET Core.

Таблица 6.1. Сводка по главе

Задача	Решение	Листинг
Создание проекта модульного тестирования	Используйте команду <code>dotnet new</code> с шаблоном проекта для предпочитаемой вами инфраструктуры тестирования	6.7
Создание теста XUnit	Создайте класс с методами, декорированными атрибутом <code>Fact</code> , и применяйте класс <code>Assert</code> для инспектирования результатов тестов <code>results</code>	6.9
Прогон модульных тестов	Используйте средства прогона тестов Visual Studio или Visual Studio Code или команду <code>dotnet test</code>	6.11
Изоляция компонентов для тестирования	Создайте имитированные реализации объектов, которые требуются для тестируемого компонента	6.12–6.19

Подготовка проекта для примера

В качестве подготовительного шага понадобится создать простой проект ASP.NET Core. Откройте окно командной подсказки PowerShell через меню Start (Пуск) в Windows, перейдите в подходящую папку и введите команды из листинга 6.1.

Совет. Проекты примеров для текущей и всех остальных глав книги доступны для загрузки по ссылке <https://github.com/apress/pro-asp.net-core-3>.

Листинг 6.1. Создание проекта для примера

```
dotnet new globaljson --sdk-version 3.1.101 --output Testing/SimpleApp
dotnet new web --no-https --output Testing/SimpleApp
    --framework netcoreapp3.1
dotnet new sln -o Testing
dotnet sln Testing add Testing/SimpleApp
```

Команды из листинга 6.1 создают новый проект по имени `SimpleApp` с применением шаблона `web`, который содержит минимальную конфигурацию для приложений ASP.NET Core. Папка проекта находится внутри папки решения под названием `Testing`.

Открытие проекта

Если вы работаете в Visual Studio, то выберите пункт меню `File⇒Open⇒Project/Solution` (Файл⇒Открыть⇒Проект/Решение), выберите файл `Testing.sln` из папки `Testing` и щелкните на кнопке `Open` (Открыть) для открытия файла решения и проекта, на который решение ссылается. Если вы используете Visual Studio Code, тогда выберите пункт меню `File⇒Open Folder` (Файл⇒Открыть папку), перейдите к папке `Testing` и щелкните на кнопке `Select Folder` (Выбрать папку).

Выбор порта HTTP

Если вы применяете Visual Studio, тогда выберите пункт меню Project⇒SimpleApp Properties (Проект⇒Свойства SimpleApp), перейдите в раздел Debug (Отладка) и измените порт HTTP на 5000 в поле App URL (URL приложения), как показано на рис. 6.1. Для сохранения нового порта выберите пункт меню File⇒Save All (Файл⇒Сохранить все). (Такое изменение не требуется, если вы используете Visual Studio Code.)

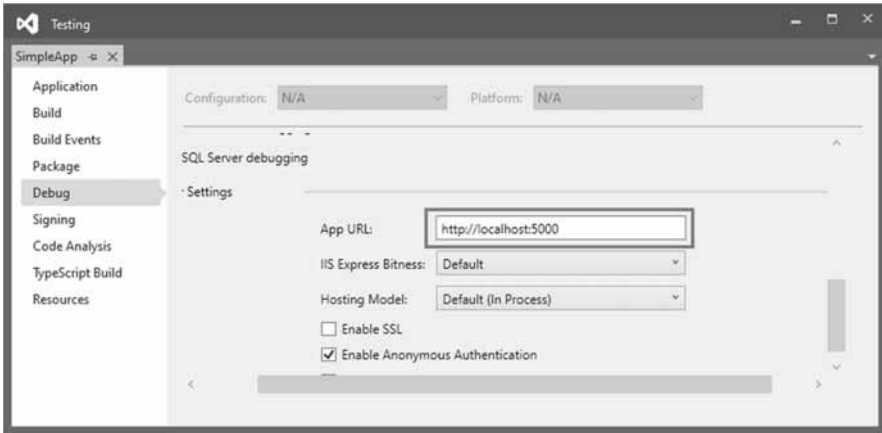


Рис. 6.1. Установка порта HTTP

Включение инфраструктуры MVC

Как объяснялось в главе 1, ASP.NET Core поддерживает разнообразные прикладные инфраструктуры, однако в этой главе я продолжаю применять MVC Framework. Другие инфраструктуры будут представлены в приложении SportsStore, построение которого начнется в главе 7, но в данный момент MVC Framework обеспечивает основу для демонстрации способа проведения модульного тестирования, знакомую по предшествующим примерам. Добавьте операторы, показанные в листинге 6.2, в файл Startup.cs из папки SimpleApp.

Листинг 6.2. Включение инфраструктуры MVC в файле Startup.cs из папки SimpleApp

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

namespace SimpleApp {
    public class Startup {
```

```

public void ConfigureServices(IServiceCollection services) {
    services.AddControllersWithViews();
}

public void Configure(IApplicationBuilder app,
                     IWebHostEnvironment env) {
    if (env.IsDevelopment()) {
        app.UseDeveloperExceptionPage();
    }

    app.UseRouting();

    app.UseEndpoints(endpoints => {
        endpoints.MapDefaultControllerRoute();
        // endpoints.MapGet("/", async context => {
        //     await context.Response.WriteAsync("Hello World!");
        // });
    });
}
}
}

```

Создание компонентов приложения

Теперь, когда инфраструктура MVC настроена, можно добавить компоненты приложения, которые будут применяться при модульном тестировании.

Создание модели данных

Давайте для начала создадим простой класс модели, чтобы иметь данные, с которыми можно будет работать. Добавьте папку по имени `Models`, создайте внутри нее файл класса `Product.cs` и поместите в него код, приведенный в листинге 6.3.

Листинг 6.3. Содержимое файла `Product.cs` из папки `SimpleApp/Models`

```

namespace SimpleApp.Models {
    public class Product {
        public string Name { get; set; }
        public decimal? Price { get; set; }

        public static Product[] GetProducts() {
            Product kayak = new Product {
                Name = "Kayak", Price = 275M
            };

            Product lifejacket = new Product {
                Name = "Lifejacket", Price = 48.95M
            };

            return new Product[] { kayak, lifejacket };
        }
    }
}

```

В классе `Product` определены свойства `Name` и `Price`, а также статический метод по имени `GetProducts`, возвращающий массив `Products`.

Создание контроллера и представления

В примере, приведенном в главе, используется простой контроллер. Создайте папку `Controllers` и добавьте в нее файл класса по имени `HomeController.cs`, содержимое которого показано в листинге 6.4.

Листинг 6.4. Содержимое файла `HomeController.cs` из папки `SimpleApp/Controllers`

```
using Microsoft.AspNetCore.Mvc;
using SimpleApp.Models;

namespace SimpleApp.Controllers {
    public class HomeController : Controller {
        public IActionResult Index() {
            return View(Product.GetProducts());
        }
    }
}
```

Метод действия `Index` сообщает ASP.NET Core о необходимости визуализации стандартного представления с передачей объектов `Product`, полученных из метода `Product.GetProducts`. Чтобы создать представление для метода действия, добавьте папку `Views/Home` (создав папку `Views` и затем внутри нее папку `Home`) и поместите в нее представление Razor по имени `Index.cshtml` с содержимым, приведенным в листинге 6.5.

Листинг 6.5. Содержимое файла `Index.cshtml` из папки `SimpleApp/Views/Home`

```
@using SimpleApp.Models
@model IEnumerable<Product>
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Simple App</title>
</head>
<body>
    <ul>
        @foreach (Product p in Model) {
            <li>Name: @p.Name, Price: @p.Price</li>
        }
    </ul>
</body>
</html>
```

Запуск примера приложения

Запустите ASP.NET Core, выбрав в меню `Debug` (Отладка) пункт `Start Without Debugging` (Запустить без отладки) для Visual Studio или пункт `Run Without Debugging` (Запустить без отладки) для Visual Studio Code либо выполнив команду из листинга 6.6 в папке `SimpleApp`.

Листинг 6.6. Запуск примера приложения

```
dotnet run
```

Запросите `http://localhost:5000`; вы увидите вывод, показанный на рис. 6.2.

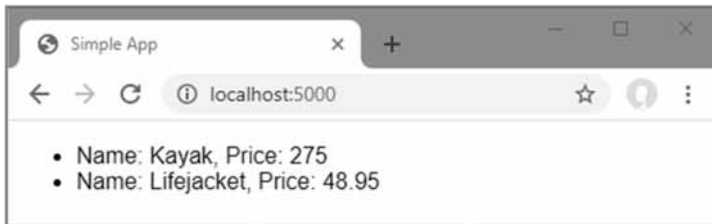


Рис. 6.2. Выполнение примера приложения

Создание проекта модульного тестирования

Для приложения ASP.NET Core обычно создается отдельный проект Visual Studio, содержащий модульные тесты, каждый из которых определяется как метод в классе C#. Применение отдельного проекта означает, что приложение можно развернуть, не развертывая одновременно тесты. Комплект .NET Core SDK включает шаблоны для модульного тестирования, которые используют три популярных инструмента тестирования, описанные в табл. 6.2.

Таблица 6.2. Инструменты для проектов модульного тестирования

Название	Описание
mstest	Этот шаблон создает проект, сконфигурированный для инфраструктуры MS Test, которая разработана Microsoft
nunit	Этот шаблон создает проект, сконфигурированный для инфраструктуры NUnit
xunit	Этот шаблон создает проект, сконфигурированный для инфраструктуры XUnit

Перечисленные инфраструктуры тестирования обладают почти одинаковыми наборами функциональных средств и отличаются только в том, как они реализованы и интегрированы в сторонние среды тестирования. Если у вас нет устоявшихся предпочтений, тогда я рекомендую начать с инфраструктуры XUnit, главным образом потому, что с ней легче всего работать.

По соглашению проекту модульного тестирования назначается имя `<ИмяПриложения>.Tests`. Находясь в папке `Testing`, выполните команды из листинга 6.7 для создания проекта тестирования XUnit под названием `SimpleApp.Tests`, добавьте его в файл решения и создайте ссылку между проектами, чтобы модульные тесты можно было применять к классам, определенным в проекте `SimpleApp`.

Листинг 6.7. Создание проекта модульного тестирования

```
dotnet new xunit -o SimpleApp.Tests --framework netcoreapp3.1
dotnet sln add SimpleApp.Tests
dotnet add SimpleApp.Tests reference SimpleApp
```

Если вы используете Visual Studio, то вам будет предложено повторно загрузить решение, что приведет к отображению нового проекта модульного тестирования в окне Solution Explorer наряду с существующим проектом. Вы можете обнаружить, что среда Visual Studio Code не компилирует новый проект. В таком случае выберите пункт меню Terminal⇒Configure Default Build Task (Терминал⇒Конфигурировать стандартную задачу компиляции), выберите в списке build и по запросу выберите .NET Core в списке сред.

Удаление стандартного тестового класса

Шаблон проекта добавляет в проект тестирования файл класса C#, который приведет к беспорядку в результатах последующих примеров. Либо удалите файл UnitTest1.cs из папки SimpleApp.Tests в окне Solution Explorer или в области Explorer, либо запустите в папке Testing команду, показанную в листинге 6.8.

Листинг 6.8. Удаление файла стандартного тестового класса

```
Remove-Item SimpleApp.Tests/UnitTest1.cs
```

Написание и прогон модульных тестов

Теперь, когда все подготовительные шаги завершены, можно приступить к написанию некоторых тестов. Первым делом добавьте в проект SimpleApp.Tests файл класса по имени ProductTests.cs с кодом, приведенным в листинге 6.9. Несмотря на простоту, класс содержит все, что требуется для начала модульного тестирования.

На заметку! В методе CanChangeProductPrice умышленно допущена ошибка, которая позже в разделе будет исправлена.

Листинг 6.9. Содержимое файла ProductTests.cs из папки SimpleApp.Tests

```
using SimpleApp.Models;
using Xunit;

namespace SimpleApp.Tests {
    public class ProductTests {
        [Fact]
        public void CanChangeProductName() {
            // Организация
            var p = new Product { Name = "Test", Price = 100M };
            // Действие
            p.Name = "New Name";
            // Утверждение
            Assert.Equal("New Name", p.Name);
        }

        [Fact]
        public void CanChangeProductPrice() {
            // Организация
            var p = new Product { Name = "Test", Price = 100M };
```

```

    // Действие
    p.Price = 200M;
    // Утверждение
    Assert.Equal(100M, p.Price);
}
}
}

```

В классе `ProductTests` присутствуют два модульных теста, проверяющих поведение класса модели `Product` из проекта `SimpleApp`. Проект тестирования может содержать много классов, которые способны включать множество модульных тестов.

По соглашению имя тестового метода описывает то, что делает тест, а имя класса — то, что подвергается тестированию. Это облегчает структурирование тестов в проекте и упрощает понимание того, какими будут результаты всех тестов, когда они прогоняются средой `Visual Studio`. Имя `ProductTests` указывает, что класс содержит тесты для класса `Product`, а имена методов говорят о том, что они проверяют возможность изменения названия и цены объекта `Product`.

Атрибут `Fact` применяется к каждому методу, указывая на то, что метод является тестом. Внутри тела метода модульный тест следует шаблону, который называется *организация/действие/утверждение* (`arrange/act/assert` — A/A/A). *Организация* относится к настройке условий для теста, *действие* — к выполнению теста, а *утверждение* — к проверке того, что результат оказался тем, который ожидали.

Разделы организации и действия этих тестов представляют собой обычный код `C#`, но раздел утверждения обрабатывается инфраструктурой `xUnit.net`, которая предоставляет класс по имени `Assert`, чьи методы используются для проверки, является ли результат действия тем, что ожидался.

Совет. Атрибут `Fact` и класс `Assert` определены в пространстве имен `Xunit`, для которого должен быть предусмотрен оператор `using` в каждом тестовом классе.

Методы класса `Assert` определены как статические и применяются для выполнения разных видов сравнений между ожидаемыми и действительными результатами. В табл. 6.3 описаны распространенные методы класса `Assert`.

Таблица 6.3. Часто используемые методы класса `Assert` из инфраструктуры `xUnit.net`

Метод	Описание
<code>Equal(expected, result)</code>	Добавляет утверждение о том, что результат равен ожидаемому исходу. Существуют перегруженные версии этого метода для сравнения различных типов и для сравнения коллекций. Имеется также версия, которая принимает дополнительный аргумент в форме объекта, реализующего интерфейс <code>IEqualityComparer<T></code> для сравнения объектов
<code>NotEqual(expected, result)</code>	Добавляет утверждение о том, что результат не равен ожидаемому исходу
<code>True(result)</code>	Добавляет утверждение о том, что результат равен <code>true</code>
<code>False(result)</code>	Добавляет утверждение о том, что результат равен <code>false</code>

Метод	Описание
IsType (expected, result)	Добавляет утверждение о том, что результат принадлежит указанному типу
IsNotType (expected, result)	Добавляет утверждение о том, что результат не принадлежит указанному типу
IsNull (result)	Добавляет утверждение о том, что результат равен null
IsNotNull (result)	Добавляет утверждение о том, что результат не равен null
InRange (result, low, high)	Добавляет утверждение о том, что результат находится между low и high
NotInRange (result, low, high)	Добавляет утверждение о том, что результат не находится между low и high
Throws (exception, expression)	Добавляет утверждение о том, что указанное выражение генерирует исключение заданного типа

Каждый метод класса `Assert` позволяет выполнять разные виды сравнений и генерирует исключение, если результат оказывается не тем, который ожидался. Исключение применяется для указания на то, что тест не прошел. В тестах из листинга 6.9 метод `Equal` использовался для определения, корректно ли изменилось значение свойства:

```
...
Assert.Equal("New Name", p.Name);
...
```

Прогон тестов с помощью проводника тестов Visual Studio

Среда Visual Studio предлагает поддержку для поиска и прогона модульных тестов посредством окна Test Explorer (Проводник тестов), которое доступно через пункт меню `Test⇒Test Explorer` (Тест⇒Проводник тестов) и показано на рис. 6.3.



Рис. 6.3. Окно Test Explorer в Visual Studio

Совет. Если вы не видите модульные тесты в окне Test Explorer, тогда скомпилируйте решение. Компиляция инициирует процесс, с помощью которого обнаруживаются модульные тесты.

Прогоните тесты, щелкнув на кнопке Run All Tests (Выполнить все тесты) в окне Test Explorer (это кнопка с изображением двух стрелок, первая в верхней части окна). Как отмечалось, тест `CanChangeProductPrice` содержит ошибку, препятствующую его успешному прохождению, что ясно видно в результатах тестирования на рис. 6.3.

Прогон тестов с помощью Visual Studio Code

Среда Visual Studio Code обнаруживает тесты и позволяет их прогонять с применением средства CodeLens, которое отображает детали о возможностях кода в редакторе. Чтобы прогнать все тесты в классе `ProductTests`, щелкните на ссылке `Run All Tests` (Выполнить все тесты) в редакторе кода, когда открыт класс модульного тестирования (рис. 6.4).

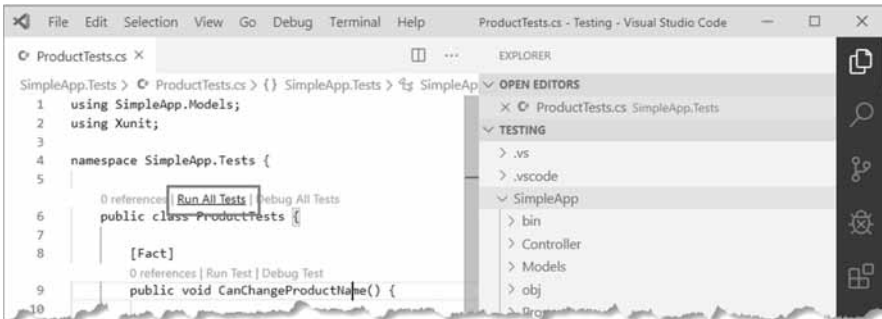


Рис. 6.4. Прогон тестов с помощью средства CodeLens в Visual Studio Code

Совет. Если вы не видите средства CodeLens, тогда закройте и снова откройте папку `Testing` в Visual Studio Code.

Среда Visual Studio Code прогоняет тесты с использованием инструментов командной строки, которые рассматриваются в следующем разделе, и результаты отображаются в виде текста в окне терминала.

Прогон тестов с помощью командной строки

Для прогона тестов в проекте выполните в папке `Testing` команду, приведенную в листинге 6.10.

Листинг 6.10. Выполнение модульных тестов

```
dotnet test
```

Тесты обнаруживаются и выполняются, производя показанные ниже результаты, которые отражают упомянутую ранее преднамеренно внесенную ошибку:

```

Test run for C:\Users\adam\SimpleApp.Tests.dll (.NETCoreApp,Version=v3.1)
Microsoft (R) Test Execution Command Line Tool Version 16.3.0
Copyright (c) Microsoft Corporation. All rights reserved.
Starting test execution, please wait...

A total of 1 test files matched the specified pattern.
[xUnit.net 00:00:00.83] SimpleApp.Tests.ProductTests.
CanChangeProductPrice [FAIL]
  X SimpleApp.Tests.ProductTests.CanChangeProductPrice [6ms]
  Error Message:
  Assert.Equal() Failure
  
```

```

Expected: 100
Actual: 200
  Stack Trace:
    at SimpleApp.Tests.ProductTests.CanChangeProductPrice()
in C:\Users\adam\Documents\
Books\Pro ASP.NET Core MVC 3\Source Code\Current\Testing\
SimpleApp.Tests\ProductTests.cs:line 31
Test Run Failed.

Total tests: 2
  Passed: 1
  Failed: 1
Total time: 1.7201 Seconds

```

Исправление модульного теста

Проблема с модульным тестом касается аргументов метода `Assert.Equal`, который сравнивает результат теста с исходным значением свойства `Price`, а не со значением, на которое оно изменилось. В листинге 6.11 проблема устранена.

Совет. Когда тест не проходит, то прежде чем просматривать компонент, на который он нацелен, всегда полезно проверить правильность самого теста, особенно если тест только что написан или недавно был модифицирован.

Листинг 6.11. Исправление теста в файле `ProductTests.cs` из папки `SimpleApp.Tests`

```

using SimpleApp.Models;
using Xunit;

namespace SimpleApp.Tests {
    public class ProductTests {
        [Fact]
        public void CanChangeProductName() {
            // Организация
            var p = new Product { Name = "Test", Price = 100M };
            // Действие
            p.Name = "New Name";
            // Утверждение
            Assert.Equal("New Name", p.Name);
        }

        [Fact]
        public void CanChangeProductPrice() {
            // Организация
            var p = new Product { Name = "Test", Price = 100M };
            // Действие
            p.Price = 200M;
            // Утверждение
            Assert.Equal(200M, p.Price);
        }
    }
}

```

Снова прогоните тесты и вы увидите, что все они проходят. Если вы применяете Visual Studio, то можете щелкнуть на кнопке Run Failed Tests (Выполнить отказавшие тесты), что приведет к выполнению только тестов, которые не прошли (рис. 6.5).

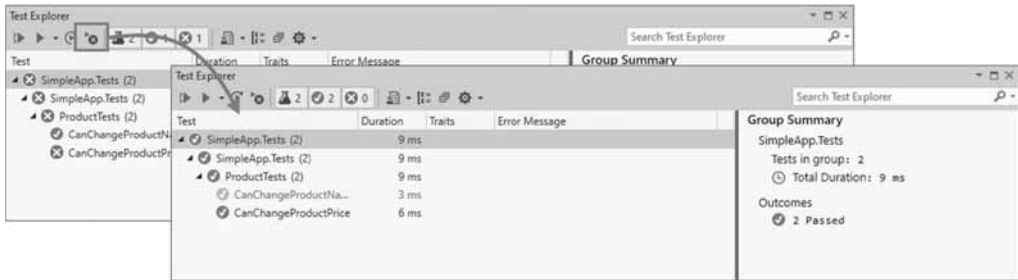


Рис. 6.5. Выполнение только тестов, которые не прошли

Изолирование компонентов для модульного тестирования

Писать модульные тесты для классов моделей вроде `Product` легко. Класс `Product` не только прост, он также автономен, т.е. при выполнении какого-то действия над объектом `Product` можно иметь уверенность в том, что тестируется функциональность, предоставляемая классом `Product`.

С другими компонентами в приложении ASP.NET Core ситуация сложнее, потому что между ними есть зависимости. Следующий набор определяемых тестов будет оперировать на контроллере, исследуя последовательность объектов `Product`, которая передается между контроллером и представлением.

При сравнении объектов, являющихся экземплярами специальных классов, необходимо использовать метод `Assert.Equal` из `xUnit.net`, который принимает аргумент, реализующий интерфейс `IEqualityComparer<T>`, так что объекты можно сравнивать. Сначала понадобится добавить в проект модульного тестирования файл класса по имени `Comparer.cs` и поместить в него определения вспомогательных классов, приведенные в листинге 6.12.

Листинг 6.12. Содержимое файла `Comparer.cs` из папки `SimpleApp.Tests`

```
using System;
using System.Collections.Generic;
namespace SimpleApp.Tests {
    public class Comparer {
        public static Comparer<U> Get<U>(Func<U, U, bool> func) {
            return new Comparer<U>(func);
        }
    }

    public class Comparer<T> : Comparer, IEqualityComparer<T> {
        private Func<T, T, bool> comparisonFunction;

        public Comparer(Func<T, T, bool> func) {
            comparisonFunction = func;
        }
    }
}
```

```

    public bool Equals(T x, T y) {
        return comparisonFunction(x, y);
    }
    public int GetHashCode(T obj) {
        return obj.GetHashCode();
    }
}
}

```

Показанные классы позволят создавать объекты реализации `IEquality Comparer<T>` с применением лямбда-выражений вместо определения нового класса для каждого типа сравнения, которое нужно предпринимать. Это не является жизненно необходимым, но упростит код в классах модульных тестов и сделает его легче для понимания и сопровождения.

Теперь, когда можно легко делать сравнения, давайте рассмотрим проблему зависимостей между компонентами в приложении. Добавьте в проект `SimpleApp.Tests` новый файл класса по имени `HomeControllerTests.cs` и поместите в него определение модульного теста, представленное в листинге 6.13.

Листинг 6.13. Содержимое файла `HomeControllerTests.cs` из папки `SimpleApp.Tests`

```

using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using SimpleApp.Controllers;
using SimpleApp.Models;
using Xunit;

namespace SimpleApp.Tests {
    public class HomeControllerTests {
        [Fact]
        public void IndexActionModelIsComplete() {
            // Организация
            var controller = new HomeController();
            Product[] products = new Product[] {
                new Product { Name = "Kayak", Price = 275M },
                new Product { Name = "Lifejacket", Price = 48.95M }
            };
            // Действие
            var model = (controller.Index() as ViewResult)?.ViewData.Model
                as IEnumerable<Product>;
            // Утверждение
            Assert.Equal(products, model,
                Comparer.Get<Product>((p1, p2) => p1.Name == p2.Name
                    && p1.Price == p2.Price));
        }
    }
}

```

Модульный тест создает массив объектов `Product` и проверяет, что они соответствуют объектам, предоставляемым методом действия `Index` в качестве модели представления. (На раздел “Действие” можно пока не обращать внимания; класс `ViewResult` объясняется в главах 21 и 22. В настоящий момент достаточно знать, что здесь получаются данные модели, возвращаемые методом действия `Index`.)

Тест успешно проходит, но это бесполезный результат, поскольку данные, участвующие в тестировании, поступают из жестко закодированных объектов класса `Product`. Например, невозможно написать тест, который позволит удостовериться в корректном поведении контроллера при наличии более двух объектов `Product` или когда свойство `Price` первого объекта первого объекта содержит дробную часть. Общий эффект заключается в том, что проводится тестирование совместного поведения классов `HomeController` и `Product`, но лишь для специфических жестко закодированных объектов.

Модульные тесты эффективны, когда они нацелены на небольшие части приложения, такие как отдельный метод или класс. Контроллер `Home` необходимо изолировать от остальной части приложения, чтобы ограничить область действия теста и исключить влияние со стороны хранилища.

Изолирование компонента

Ключом к изолированию компонентов является использование интерфейсов C#. Чтобы отделить контроллер от хранилища, добавьте в папку `Models` новый файл по имени `IDataSource.cs` с определением интерфейса, показанным в листинге 6.14.

Листинг 6.14. Содержимое файла `IDataSource.cs` из папки `SimpleApp/Models`

```
using System.Collections.Generic;

namespace SimpleApp.Models {
    public interface IDataSource {
        IEnumerable<Product> Products { get; }
    }
}
```

В листинге 6.15 из класса `Product` удален статический метод и создан новый класс, который реализует интерфейс `IDataSource`.

Листинг 6.15. Создание источника данных в файле `Product.cs` из папки `SimpleApp/Models`

```
using System.Collections.Generic;

namespace SimpleApp.Models {
    public class Product {
        public string Name { get; set; }
        public decimal? Price { get; set; }
    }

    public class ProductDataSource : IDataSource {
        public IEnumerable<Product> Products =>
            new Product[] {
                new Product { Name = "Kayak", Price = 275M },
                new Product { Name = "Lifejacket", Price = 48.95M }
            };
    }
}
```

Следующий шаг предусматривает модификацию контроллера, чтобы применять в качестве источника данных класс `ProductDataSource` (листинг 6.16).

Совет. Инфраструктура ASP.NET Core поддерживает более элегантный подход к решению этой проблемы, который называется *внедрением зависимостей* и описан в главе 14. Внедрение зависимостей часто вызывает путаницу и потому в настоящей главе компоненты изолируются более простым и ручным способом.

Листинг 6.16. Добавление свойства в файле HomeController.cs из папки SimpleApp/Controllers

```
using Microsoft.AspNetCore.Mvc;
using SimpleApp.Models;

namespace SimpleApp.Controllers {
    public class HomeController : Controller {
        public IDataSource dataSource = new ProductDataSource();
        public IActionResult Index() {
            return View(dataSource.Products);
        }
    }
}
```

Модификация может выглядеть незначительной, но она позволяет изменять источник данных, который контроллер применяет во время тестирования, что демонстрирует способ изоляции контроллера. В листинге 6.17 модульные тесты контроллера обновлены, чтобы использовать специальную версию хранилища.

Листинг 6.17. Изоляция контроллера в файле HomeControllerTests.cs из папки SimpleApp.Tests

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using SimpleApp.Controllers;
using SimpleApp.Models;
using Xunit;

namespace SimpleApp.Tests {
    public class HomeControllerTests {
        class FakeDataSource : IDataSource {
            public FakeDataSource(Product[] data) => Products = data;
            public IEnumerable<Product> Products { get; set; }
        }

        [Fact]
        public void IndexActionModelIsComplete() {
            // Организация
            Product[] testData = new Product[] {
                new Product { Name = "P1", Price = 75.10M },
                new Product { Name = "P2", Price = 120M },
                new Product { Name = "P3", Price = 110M }
            };
            IDataSource data = new FakeDataSource(testData);
            var controller = new HomeController();
            controller.dataSource = data;
            // Действие
            var model = (controller.Index() as IActionResult)?.ViewData.Model
                as IEnumerable<Product>;
        }
    }
}
```

```

// Утверждение
Assert.Equal(data.Products, model,
    Comparer.Get<Product>((p1, p2) => p1.Name == p2.Name
        && p1.Price == p2.Price));
    }
}
}

```

Здесь определена фиктивная реализация интерфейса `IDataSource`, которая позволяет использовать с контроллером любые тестовые данные.

Понятие разработки через тестирование

В настоящей главе я придерживаюсь наиболее широко применяемого стиля модульного тестирования, при котором мы пишем функцию приложения и затем тестируем ее, чтобы удостовериться в том, что она работает требуемым образом. Такой стиль популярен, поскольку большинство разработчиков думают сначала о прикладном коде, а затем о его тестировании (я определенно подпадаю под эту категорию).

Проблема такого подхода в том, что он склоняет к созданию модульных тестов только для того прикладного кода, который было трудно писать или сложно отлаживать, оставляя ряд аспектов каких-то функций лишь частично протестированными или вообще без тестирования.

Альтернативным подходом является *разработка через тестирование* (Test-Driven Development — TDD). Существует много вариаций TDD, но основная идея в том, что тесты для функции пишутся до того, как она реализуется. Написание тестов первыми заставляет более тщательно думать о реализуемой спецификации и о том, как узнать, что функция была реализована корректно. Вместо погружения в детали реализации подход TDD заставляет заранее продумывать, чем будет измеряться успех или неудача.

Все создаваемые тесты изначально не будут проходить, т.к. новая функция еще не реализована. Однако по мере добавления кода в приложение тесты постепенно будут переходить из красного состояния в зеленое, и ко времени завершения функции все тесты окажутся пройденными. Подход TDD требует дисциплины, но приводит к получению более полного набора тестов и может дать в результате более надежный и устойчивый код.

Использование инфраструктуры имитации

Создать фиктивную реализацию интерфейса `IDataSource` было просто, но большинство классов, для которых требуются фиктивные реализации, сложнее и не могут поддерживаться с той же легкостью.

Более эффективный подход предусматривает применение пакета имитации, который облегчает создание фиктивных — или имитированных — объектов для тестов. Доступно много пакетов имитации, но на протяжении долгих лет я использую один из них, который называется `Moq`. Чтобы добавить пакет `Moq` в проект тестирования, запустите в папке `Testing` команду из листинга 6.18.

На заметку! Пакет `Moq` добавляется в проект модульного тестирования, но не в проект, который содержит приложение, подлежащее тестированию.

Листинг 6.18. Установка пакета имитации

```
dotnet add SimpleApp.Tests package Moq --version 4.13.1
```

Создание имитированного объекта

Инфраструктура Moq может применяться для создания фиктивного объекта реализации интерфейса `IDataSource` без необходимости в определении специального тестового класса (листинг 6.19).

Листинг 6.19. Создание имитированного объекта в файле `HomeControllerTests.cs` из папки `SimpleApp.Tests`

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using SimpleApp.Controllers;
using SimpleApp.Models;
using Xunit;
using Moq;
namespace SimpleApp.Tests {
    public class HomeControllerTests {
        // class FakeDataSource : IDataSource {
        //     public FakeDataSource(params Product[] data) => Products = data;
        //     public IEnumerable<Product> Products { get; set; }
        // }
        [Fact]
        public void IndexActionModelIsComplete() {
            // Организация
            Product[] testData = new Product[] {
                new Product { Name = "P1", Price = 75.10M },
                new Product { Name = "P2", Price = 120M },
                new Product { Name = "P3", Price = 110M }
            };
            var mock = new Mock<IDataSource>();
            mock.SetupGet(m => m.Products).Returns(testData);
            var controller = new HomeController();
            controller.dataSource = mock.Object;
            // Действие
            var model = (controller.Index() as ViewResult)?.ViewData.Model
                as IEnumerable<Product>;
            // Утверждение
            Assert.Equal(testData, model,
                Comparer.Get<Product>((p1, p2) => p1.Name == p2.Name
                    && p1.Price == p2.Price));
            mock.VerifyGet(m => m.Products, Times.Once);
        }
    }
}
```

Использование Moq позволило удалить фиктивные реализации интерфейса `IRepository` и заменить их несколькими строками кода. Здесь не приводятся детальные сведения о разных поддерживаемых Moq средствах, но на примерах объясняется способ применения Moq. (Примеры и документация по Moq доступны по адресу <https://github.com/Moq/moq4>. При обсуждении модульного тестирования различ-

ных типов компонентов в оставшихся главах книги также будут приводиться примеры.) Первым делом создается новый объект `Mock` с указанием интерфейса, который должен быть реализован:

```
...
var mock = new Mock<IDataSource>();
...
```

Созданный объект `Mock` будет имитировать интерфейс `IDataSource`. Чтобы создать реализацию свойства `Product`, применяется метод `SetupGet`:

```
...
mock.SetupGet(m => m.Products).Returns(testData);
...
```

Метод `SetupGet` используется для реализации средства извлечения для свойства. Аргументом этого метода является лямбда-выражение, которое указывает подлежащее реализации свойство (`Products` в данном примере). Метод `Returns` вызывается на результате, полученном из метода `SetupGet`, чтобы указать результат, который будет возвращаться при чтении значения свойства. В классе `Mock` определено свойство `Object`, возвращающее объект, который реализует указанный интерфейс и обладает ранее определенным поведением. Свойство `Object` применяется для установки поля `dataSource`, определенного в классе `HomeController`:

```
...
controller.dataSource = mock.Object;
...
```

Последним использованным средством `Mock` была проверка того, что к свойству `Products` происходило только одно обращение:

```
...
mock.VerifyGet(m => m.Products, Times.Once);
...
```

Метод `VerifyGet` относится к тем методам класса `Mock`, которые инспектируют состояние имитированного объекта, когда тест завершен. В данном случае метод `VerifyGet` позволяет проверить, сколько раз читалось свойство `Products`. Значение `Times.Once` указывает, что метод `VerifyGet` должен генерировать исключение, если свойство читалось не в точности один раз, и это приведет к тому, что тест не пройдет. (Методы класса `Assert`, обычно применяемые в тестах, генерируют исключение, когда тест не проходит, и потому при работе с имитированными объектами метод `VerifyGet` можно использовать для замены метода класса `Assert`.)

Общий эффект оказывается таким же, как тот, что обеспечивала фиктивная реализация, но имитация является более гибкой и лаконичной и может дать больше сведений о поведении тестируемых компонентов.

Резюме

Глава была сконцентрирована на модульном тестировании, которое способно стать мощным инструментом, направленным на повышение качества кода. Модульное тестирование не подойдет абсолютно каждому разработчику, но с ним полезно поэкспериментировать, и оно может быть полезным, даже если применяется только для сложных функций или для диагностики проблем. Было описано использование инфраструктуры тестирования `xUnit.net`, объяснена важность изоляции компонентов в целях тестирования и продемонстрированы некоторые инструменты и методики, позволяющие упростить код модульных тестов. В следующей главе начнется разработка более реалистичного проекта под названием `SportsStore`.