Содержание

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 `dotnet new` с шаблоном проекта для предпочитаемой вами инфраструктуры тестирования	6.7
Создание теста XUnit	Создайте класс с методами, декорированными атрибутом `Fact`, и применяйте класс `Assert` для инспектирования результатов тестов results	6.9
Прогон модульных тестов	Используйте средства прогона тестов Visual Studio или Visual Studio Code или команду `dotnet test`	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.)

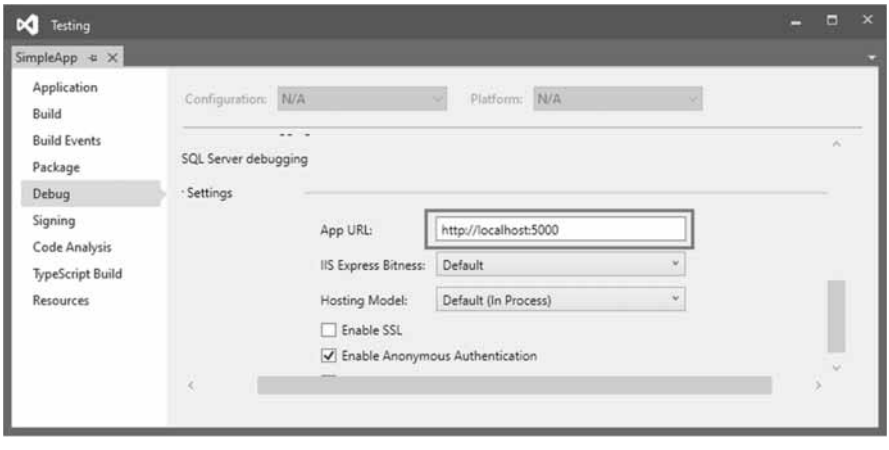

Рис. 6.1. Установка порта HTTP

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 ViewResult 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.

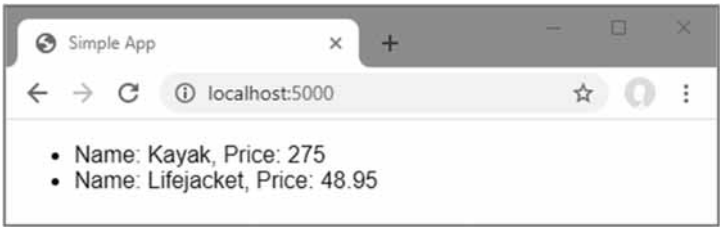

Рис. 6.2. Выполнение примера приложения

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 приступать 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 `Asset` 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
`Equal(expected, result)`	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 `IEqualityComparer<T>` для сравнения объектов
`NotEqual(expected, result)`	Добавляет утверждение о том, что результат не равен ожидаемому исходу
`True(result)`	Добавляет утверждение о том, что результат равен `true`
`False(result)`	Добавляет утверждение о том, что результат равен `false`

Окончание табл. 6.3

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.

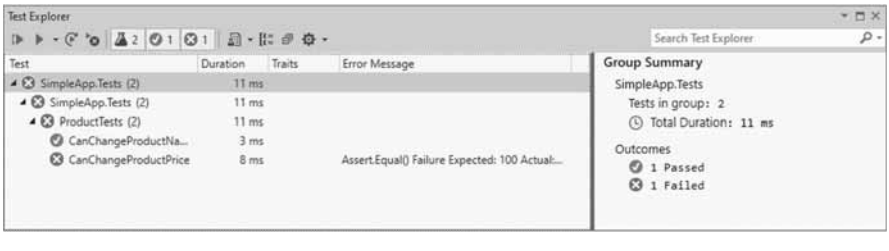

Рис. 6.3. Окно Test Explorer в Visual Studio

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).

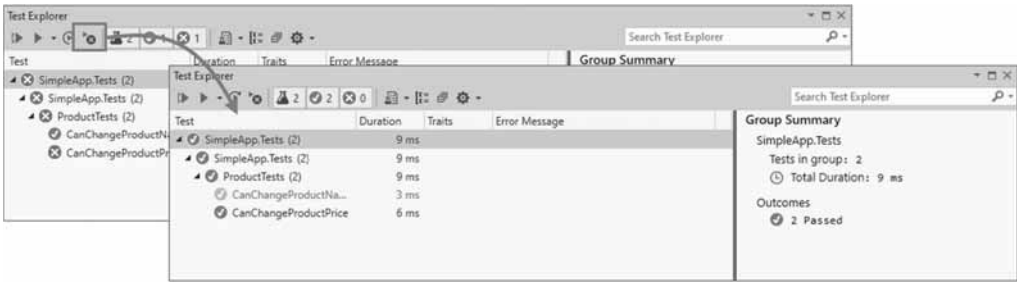

Рис. 6.5. Выполнение только тестов, которые не прошли

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 `IEqualityComparer<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 ViewResult 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 ViewResult)?.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 Moq 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.