ГЛАВА 5

Работа с Razor

Razor — это название механизма визуализации, который был введен Microsoft в версии MVC 3 и переработан в версии MVC 4 (хотя изменения были относительно небольшими). Механизм визуализации обрабатывает контент ASP.NET и ищет инструкции, которые обычно вставляют динамический контент в вывод, отправляемый браузеру. В Microsoft поддерживаются два механизма визуализации: механизм ASPX, работающий с дескрипторами <% и %>, которые были основной опорой разработки ASP.NET в течение многих лет, и механизм Razor, имеющий дело с областями контента, которые обозначены с помощью символа @.

По большому счету, если вы знакомы с синтаксисом <% %>, то не будете иметь проблем с освоением Razor, хотя в этом механизме присутствует несколько новых правил. В настоящей главе мы представим краткий экскурс в синтаксис Razor, так что вы сумеете опознать новые элементы, как только с ними столкнетесь. Мы не собираемся превращать эту главу в исчерпывающий справочник по Razor; скорее, ее можно считать ускоренным курсом по синтаксису. Особенности Razor будут раскрываться в последующих главах книги.

Совет. Механизм Razor тесно связан с разработкой MVC, но после появления ASP.NET 4.5 механизм визуализации Razor поддерживает также и разработку ASP.NET Web Pages.

Создание проекта для примера

Для демонстрации возможностей и синтаксиса Razor мы создали в Visual Studio новый проект ASP.NET MVC 4 Web Application (Веб-приложение ASP.NET MVC 4) и выбрали для него шаблон Empty (Пустой).

Определение модели

Мы собираемся использовать очень простую модель предметной области и воспроизвести тот же самый класс предметной области `Product`, который использовался в предыдущей главе. Добавьте в папку `Models` файл класса по имени `Product.cs` с контентом, показанным в листинге 5.1.

Листинг 5.1. Создание простого класса модели предметной области

```
namespace Razor.Models {

    public class Product {

        public int ProductID { get; set; }
        public string Name { get; set; }
        public string Description { get; set; }
```

```
        public decimal Price { get; set; }
        public string Category { set; get; }
    }
}
```

Определение контроллера

Чтобы добавить в проект контроллер, щелкните правой кнопкой мыши на папке `Controllers` в своем проекте и выберите в контекстном меню пункт Add (Добавить), а затем Controller (Контроллер). Укажите в качестве имени `HomeController` и выберите в списке Template (Шаблон) вариант Empty MVC Controller (Пустой контроллер MVC), как показано на рис. 5.1.

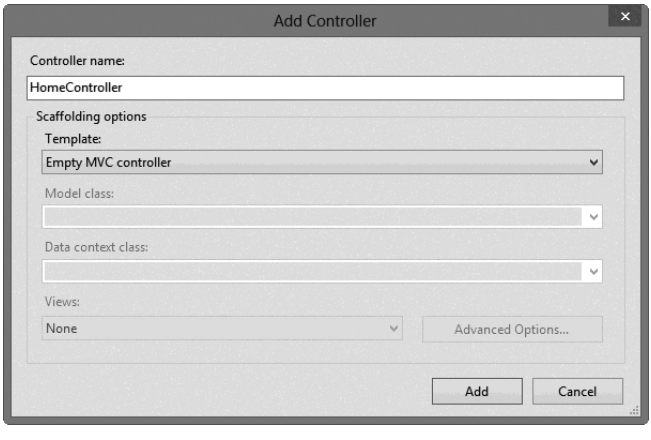

Рис. 5.1. Создание контроллера `ProductController`

Щелкните на кнопке Add (Добавить) для создания класса `Controller`, после чего отредактируйте контент файла, чтобы он соответствовал показанному в листинге 5.2.

Листинг 5.2. Простой контроллер

```
using Razor.Models;
using System;
using System.Collections.Generic;
using System.Web.Mvc;

namespace Razor.Controllers {
    public class HomeController : Controller {
        Product myProduct = new Product {
            ProductID = 1,
            Name = "Kayak",
            Description = "A boat for one person",
            Category = "Watersports",
            Price = 275M
        };

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

Мы определили метод действия по имени Index, в котором создали объект Product и заполнили его свойства. Объект Product передается методу View, поэтому он используется в качестве модели, когда представление визуализируется. При вызове метода View имя файла представления не указывается, поэтому будет выбрано стандартное представление для метода действия (файл представления создается следующим).

Создание представления

Чтобы создать представление, щелкните правой кнопкой мыши на методе Index класса HomeController и выберите в контекстном меню пункт Add View (Добавить представление). Отметьте флажок Create a strongly-typed view (Создать строго типизированное представление) и выберите класс Product в раскрывающемся списке Model class (Класс модели), как показано на рис. 5.2.

На заметку! Если вы не видите класса Product в раскрывающемся списке, скомпилируйте проект и попробуйте создать представление заново. Среда Visual Studio не распознает классы моделей до тех пор, пока они не будут скомпилированы.

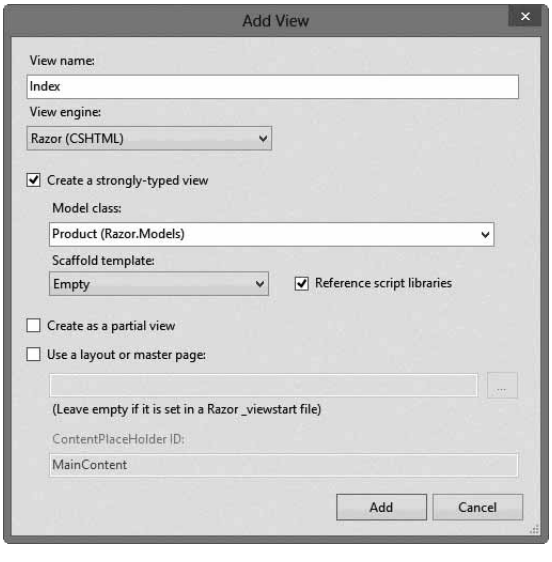

Рис. 5.2. Добавление представления Index

Удостоверьтесь, что флажок Use a layout or master page (Использовать компоновку или мастер-страницу) не отмечен. Щелкните на кнопке Add (Добавить) для создания представления, которое появится в виде файла Index.cshtml в папке Views/Product. Этот файл представления открывается для редактирования, и вы увидите, что в нем содержится тот же базовый контент, как и при создании представления в предыдущей главе (листинг 5.3).

Листинг 5.3. Простое представление Razor

```
@model Razor.Models.Product

@{
    Layout = null;
}
```

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>

    </div>
</body>
</html>
```

В последующих разделах мы рассмотрим разнообразные аспекты представления Razor и продемонстрируем различные вещи, которые с ним можно делать.

При изучении Razor полезно помнить, что представления существуют для выражения пользователю одной или более частей модели — и это означает генерацию HTML-разметки, которая отображает данные, извлеченные из одного или множества объектов. Если не забывать, что мы всегда пытаемся построить HTML-страницу, которая может быть отправлена клиенту, то вся активность механизма Razor начинает обретать смысл.

На заметку! Ниже будет повторяться часть сведений, которые уже рассматривались в главе 2. Это сделано для того, чтобы обеспечить единственное место для справки по всем функциональным возможностям Razor.

Работа с объектом модели

Давайте начнем с самой первой строки в представлении:

```
@model Razor.Models.Product
...
```

Операторы Razor начинаются с символа @. В этом случае оператор `@model` объявляет тип объекта модели, который будет передаваться представлению из метода действия. Это позволяет ссылаться на методы, поля и свойства объекта модели представления с помощью `@Model`, как показано в листинге 5.4, где демонстрируется простое дополнение к представлению `Index`.

Листинг 5.4. Ссылка на свойство объекта модели представления в представлении Razor

```
@model Razor.Models.Product

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>
        @Model.Name
    </div>
</body>
</html>
```

На заметку! Обратите внимание, что тип объекта модели представления объявляется с использованием `@model` (со строчной буквой "m"), а доступ к свойству `Name` производится с применением `@Model` (с прописной буквой "M"). Поначалу это может слегка запутывать, но со временем становится вполне привычным.

Запустив проект, вы увидите вывод, показанный на рис. 5.3. Переходить по какому-то специфичному URL не требуется, т.к. по стандартному соглашению в проекте MVC запрос корневого URL (`/`) направляется методу действия `Index` в контроллере `Home`; в главе 13 будет продемонстрировано, как это изменить.

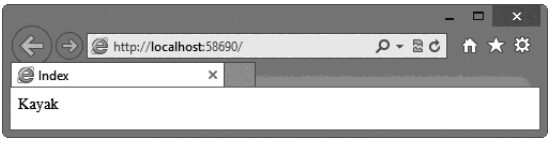

Рис. 5.3. Результат чтения значение свойства в представлении

С помощью выражения `@model` мы сообщаем MVC разновидность объекта, с которым будет осуществляться работа, а среда Visual Studio использует эту информацию несколькими способами. Прежде всего, при написании кода представления, когда вводится `@model` с последующей точкой, Visual Studio будет открывать окно с предполагаемыми именами членов, как показано на рис. 5.4. Это очень похоже на работу автозавершения лямбда-выражений, передаваемых вспомогательным методам HTML.

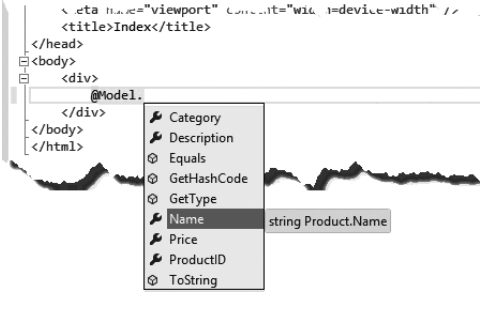

Рис. 5.4. Среда Visual Studio предлагает предположительные варианты имен членов на основе выражения `@Model`

Не менее полезно и то, что Visual Studio будет помечать ошибки, когда имеются проблемы с членами объекта модели представления, на которые производятся ссылка. Пример этого можно видеть на рис. 5.5, где производится попытка ссылки на метод `@Model.NotARealProperty`. Среде Visual Studio известно, что класс `Product`, указанный в качестве типа модели, не содержит такого свойства, поэтому ошибка подсвечивается в редакторе.

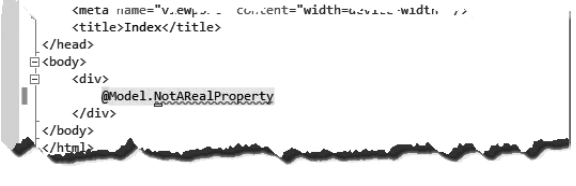

Рис. 5.5. Среда Visual Studio сообщает о проблеме с выражением `@Model`

Работа с компоновками

Вот еще одно выражение Razor из файла представления `Index.cshtml`:

```
...
@{
    Layout = null;
}
...
```

Это пример *блока кода* Razor, который позволяет включать в представление операторы C#. Блок кода открывается посредством `@{` и закрывается с помощью `}`, а содержащиеся в нем операторы оцениваются при визуализации представления.

Показанный выше блок кода устанавливает значение свойства `Layout` в `null`. Как будет объясняться в главе 18, представления Razor компилируются в классы C# приложения MVC и в базовом классе, который они используют, определено свойство `Layout`. Там же, в главе 18, будет показано, как все это работает, а пока следует знать, что результатом установки свойства `Layout` в `null` является уведомление инфраструктуры MVC о том, что наше представление является самодостаточным, и оно будет визуализировать весь свой контент, который необходимо возвратить клиенту.

Самодостаточные представления хороши для простых примеров приложений, но реальный проект может иметь десятки представлений. Компоновки на самом деле являются шаблонами, которые содержат разметку, используемую для обеспечения согласованности в рамках веб-сети — это может служить для гарантированного включения в результат правильных библиотек JavaScript или для поддержания общего внешнего вида и поведения по всему приложению.

Создание компоновки

Чтобы создать компоновку, щелкните правой кнопкой мыши на папке `Views` в окне Solution Explorer, выберите в контекстном меню пункт Add⇨New Item (Добавить⇨Новый элемент) и укажите шаблон MVC 4 Layout Page (Razor) (Страница компоновки MVC 4 (Razor)), как показано на рис. 5.6.

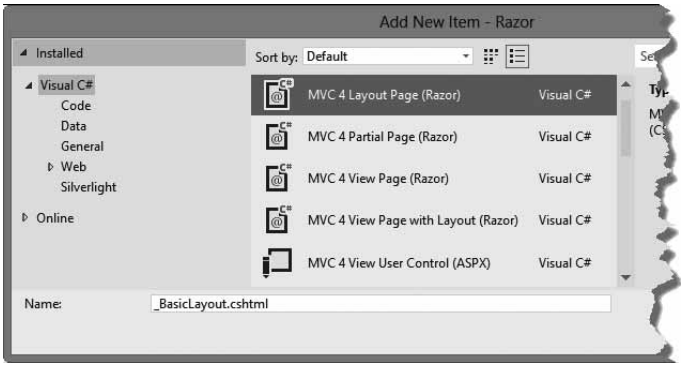

Рис. 5.6. Создание новой компоновки

Введите `_BasicLayout.cshtml` в поле Name (Имя) и щелкните на кнопке Add (Добавить) для создания файла. Сгенерированный Visual Studio для этого файла контент приведен в листинге 5.5.

На заметку! Файлы в папке `Views`, имена которых начинаются с символа подчеркивания (_), не возвращаются пользователю. С помощью таких имен файлов можно различать представления, которые должны визуализироваться, и файлы, предназначенные для их поддержки. Компоновки, являющиеся файлами поддержки, снабжаются в своих именах файлов префиксом в виде подчеркивания.

Листинг 5.5. Начальный контент компоновки

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
</head>
<body>
    <div>
        @RenderBody()
    </div>
</body>
</html>
```

Компоновки — это специализированная форма представлений; выражения @ в листинге 5.5 выделены полужирным. Вызов метода `@RenderBody` вставляет в разметку компоновки контент представления, указанный методом действия. Другое выражение Razor в компоновке обращается к свойству по имени `Title` в объекте `ViewBag`, чтобы установить контент элемента `title`.

Любые элементы в компоновке будут применяться к любому представлению, которое использует эту компоновку, и именно поэтому компоновки являются, в сущности, шаблонами. Чтобы продемонстрировать, как это работает, в листинге 5.6 добавлена простая разметка.

Листинг 5.6. Добавление элементов в компоновку

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
</head>
<body>
    <h1>Product Information</h1>
    <div style="padding: 20px; border: solid medium black; font-size: 20pt">
        @RenderBody()
    </div>
    <h2>Visit <a href="http://apress.com">Apress</a></h2>
</body>
</html>
```

Здесь была добавлена пара элементов заголовка и применены стили CSS к элементу `div`, содержащему выражение `@RenderBody`. Это сделано просто для прояснения, какой контент поступает из компоновки, а какой — из представления.

Применение компоновки

Чтобы применить компоновку к представлению, нужно всего лишь установить значение свойства `Layout`. Можно также удалить элементы, формирующие структуру завершенной HTML-страницы, поскольку они будут поступать из компоновки.

В листинге 5.7 демонстрируется применение компоновки, что существенно упрощает контент файла `Index.cshtml`.

Совет. Здесь также установлено значение свойства `ViewBag.Title`, которое будет использоваться в качестве контента для элемента `title` в HTML-документе, отправляемом обратно пользователю — это необязательная, однако хорошая практика. Если значение этого свойства не задано, инфраструктура MVC возвратит пустой элемент `title`.

Листинг 5.7. Использование свойства `Layout` для указания файла компоновки

```
@model Razor.Models.Product

@{
    ViewBag.Title = "Product Name";
    Layout = "~/Views/_BasicLayout.cshtml";
}

Product Name: @Model.Name
```

Даже для такого простого представления трансформация весьма значительна. Остался лишь код, сосредоточенный на отображении пользователю данных из объекта модели представления, а вот структура HTML-документа утрачена. Применение компоновок дает много преимуществ. Они позволяют упростить представления (как было показано в листинге), дают возможность создавать общую HTML-разметку, которую можно применять к множеству представлений, и облегчают сопровождение, поскольку общую HTML-разметку можно изменить в одном месте, зная, что изменения будут применены везде, где эта компоновка используется. Чтобы увидеть работу компоновки, запустите пример приложения. Результаты показаны на рис. 5.7.

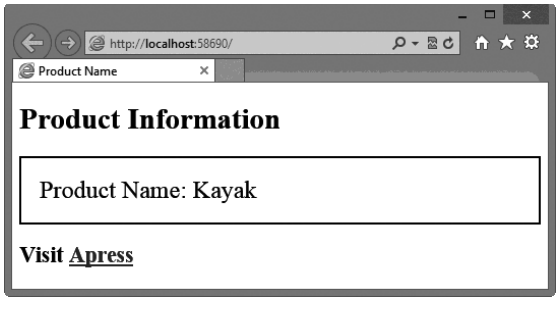

Рис. 5.7. Результат применения простой компоновки к представлению

Использование файла запуска представления

Осталась еще одна небольшая проблема, с которой необходимо разобраться — мы должны указать файл компоновки для применения в каждом представлении. Это означает, что в случае переименования файла компоновки понадобится найти все ссылающиеся на него представления и внести изменение; такой процесс чреват ошибками и противоречит основной концепции инфраструктуры MVC — легкости сопровождения.

Решить указанную проблему можно с использованием файла запуска представления. При визуализации представления инфраструктура MVC ищет файл с именем _ViewStart.cshtml. Контент этого файла будет трактоваться так, если бы он содержался в самом файле представления, и эту возможность можно применять для автоматической установки значения свойства Layout.

Чтобы создать файл запуска представления, добавьте в папку Views новый файл представления, как это объяснялось ранее. Укажите _ViewStart.cshtml в качестве имени файла и приведите контент файла в соответствие с листингом 5.8.

Листинг 5.8. Создание файла запуска представления

```
@{
    Layout = "~/Views/_BasicLayout.cshtml";
}
```

Наш файл запуска представления содержит значение для свойства Layout, а это означает, что можно удалить соответствующий оператор из файла Index.cshtml, как показано в листинге 5.9.

Листинг 5.9. Обновление представления в случае использования файла запуска представления

```
@model Razor.Models.Product

@{
    ViewBag.Title = "Product Name";
}

Product Name: @Model.Name
```

Специально указывать, что должен использоваться файл запуска представления, не требуется. Инфраструктура MVC автоматически обнаруживает данный файл и пользуется его контентом. Значениям, определяемым в файле представления, отдается преимущество, и это позволяет легко переопределять файл запуска представления.

Внимание! Важно понимать разницу между удалением свойства Layout из файла представления и его установкой в null. Если представление является самодостаточным и применять компоновку не нужно, установите свойство Layout в null. Если просто не указывать свойство Layout, инфраструктура MVC предполагает, что компоновка необходима, и должно использоваться значение, найденное в файле запуска представления.

Демонстрация разделяемых компоновок

В качестве быстрой и простой демонстрации совместного использования компоновок мы добавили к контроллеру Home новый метод действия по имени NameAndPrice. Определение этого метода приведено в листинге 5.10, содержащем измененный контент файла /Controllers/HomeController.cs.

Листинг 5.10. Добавление к контроллеру Home нового метода действия

```
using Razor.Models;
using System;
using System.Web.Mvc;

namespace Razor.Controllers {
```

```
    public class HomeController : Controller {
        Product myProduct = new Product {
            ProductID = 1,
            Name = "Kayak",
            Description = "A boat for one person",
            Category = "Watersports",
            Price = 275M
        };

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

        public ActionResult NameAndPrice() {
            return View(myProduct);
        }
    }
}
```

Этот метод действия просто передает объект `myProduct` методу представления, в точности как это делает метод действия `Index`; подобное вряд ли встретится в реальном проекте, но для демонстрации функциональности Razor вполне подойдет и такой очень простой пример.

Щелкните правой кнопкой мыши на методе `NameAndPrice` в редакторе и выберите в контекстном меню пункт Add View (Добавить представление), чтобы открыть диалоговое окно Add View (Добавление представления). Отметьте флажок Create a strongly-typed view (Создать строго типизированное представление) и выберите класс `Product` в раскрывающемся списке Model class (Класс модели). Отметьте флажок Use a layout or master page (Использовать компоновку или мастер-страницу), как показано на рис. 5.8.

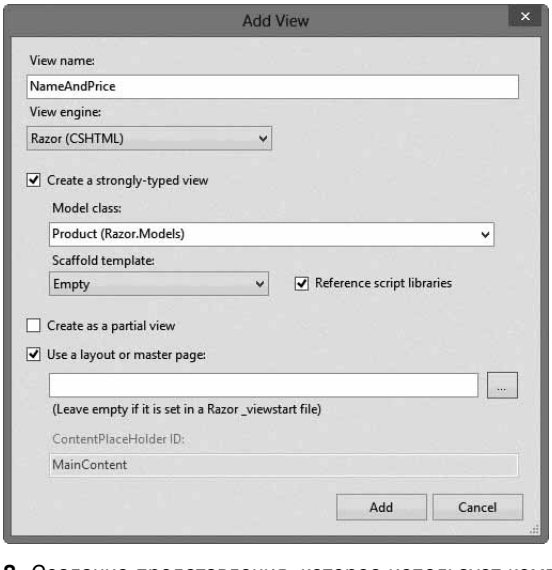

Рис. 5.8. Создание представления, которое использует компоновку

Обратите внимание на текст под флажком Use a layout or master page. Он гласит, что вы должны оставить поле пустым, если используемое представление было указано в файле запуска представления. Если теперь щелкнуть на кнопке Add (Добавить),

представление будет создано без оператора C#, устанавливающего значение свойства `Layout`. Мы собираемся явно указать представление, поэтому щелкните на кнопке с троеточием (...) справа от текстового поля. Среда Visual Studio отобразит диалоговое окно, которое позволяет выбрать файл компоновки (рис. 5.9).

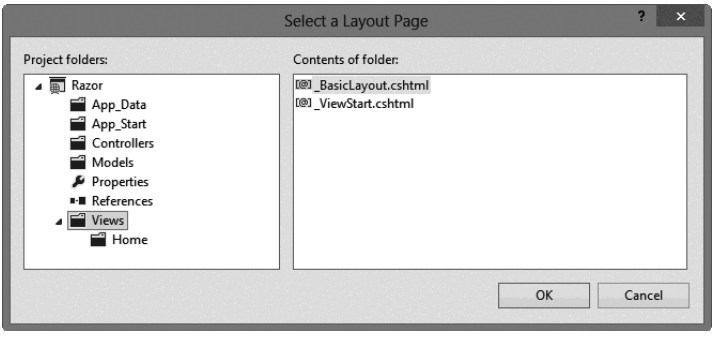

Рис. 5.9. Выбор файла компоновки

По соглашению, принятому для проекта MVC, файлы компоновки помещаются в папку `Views`, поэтому диалоговое окно на рис. 5.9 показывает контент данной папки, в котором можно сделать выбор. Тем не менее, это всего лишь соглашение, и левая панель окна позволяет осуществлять навигацию в рамках проекта на тот случай, если вы решили не соблюдать указанное соглашение.

Мы определили только один файл компоновки, поэтому выберите `_BasicLayout.cshtml` и щелкните на кнопке OK для возврата в диалоговое окно Add View. Вы увидите, что имя файла компоновки появилось в текстовом поле (рис. 5.10).

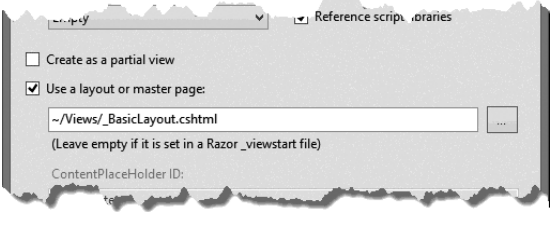

Рис. 5.10. Указание файла компоновки при создании представления

Щелкните на кнопке Add для создания файла `/Views/Home/NameAndPrice.cshtml`. Контент это файла приведен в листинге 5.11.

Листинг 5.11. Контент представления NameAndPrice

```
@model Razor.Models.Product

@{
    ViewBag.Title = "NameAndPrice";
    Layout = "~/Views/_BasicLayout.cshtml";
}

<h2>NameAndPrice</h2>
```

В случае указания компоновки среда Visual Studio применяет слегка отличающийся стандартный контент для файлов представлений, но они содержат те же самые выра-

жения Razor, которые мы используем, когда самостоятельно применяем компоновку к представлению. В завершение этого примера в листинге 5.12 показано простое добавление к файлу NameAndPrice.cshtml, которое отображает значения данных из объекта модели представления.

Листинг 5.12. Добавление к компоновке NameAndPrice

```
@model Razor.Models.Product

@{
    ViewBag.Title = "NameAndPrice";
    Layout = "~/Views/_BasicLayout.cshtml";
}

<h2>NameAndPrice</h2>
The product name is @Model.Name and it costs $@Model.Price
```

Если вы запустите приложение и перейдете на URL вида /Home/NameAndPrice, то получите результаты, показанные на рис. 5.11. Как и ожидалось, общие элементы и стили, определенные в компоновке, были применены к представлению, что демонстрирует возможность использования компоновки для создания общего внешнего вида (несмотря на его простоту и непривлекательность в данном примере).

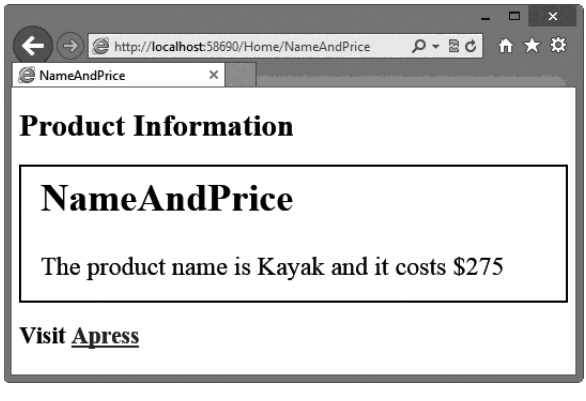

Рис. 5.11. Контент файла компоновки был применен к представлению NameAndPrice

На заметку! Мы получили бы тот же самый результат, если бы оставили текстовое поле в диалоговом окне Add View пустым и понадеялись на файл запуска представления. Файл компоновки был указан явно только для того, чтобы продемонстрировать средство Visual Studio, помогающее сделать выбор.

Использование выражений Razor

Теперь, когда были показаны основы представлений и компоновок, мы собираемся переключиться на другие виды выражений, поддерживаемые Razor, и продемонстрировать их применение для создания контента представлений.

В хорошем приложении MVC Framework имеется четкое разделение между ролями метода действия и представления. Для целей этой главы правила просты; они кратко описаны в табл. 5.1.

Таблица 5.1. Роли, исполняемые методом действия и представлением

Компонент	Что делает	Что не делает
Метод действия	Передает представлению объект модели представления	Передает представлению сформатированные данные
Представление	Использует объект модели представления для отображения контента пользователю	Изменяет любой аспект объекта модели представления

Далее в книге мы будем неоднократно возвращаться к этой теме. Чтобы извлечь максимум из MVC Framework, нужно обеспечить разделение между разными частями приложения. Вы увидите, что механизм Razor позволяет делать многое, включая использование операторов C#. Однако вы не должны применять Razor для выполнения бизнес-логики или манипулирования объектами моделей предметной области.

Кроме того, вы не должны форматировать данные, которые метод действия передает представлению. Вместо этого предоставьте возможность представлению вычислить данные, которые ему необходимо отобразить. Очень простой пример был приведен в предыдущем разделе этой главы. Мы определили метод действия по имени `NameAndPrice`, который отображает значения свойств `Name` и `Price` объекта `Product`. Несмотря на то что известно, значения каких свойств должны отображаться, модели представления передается полный объект `Product`:

```
...
public ActionResult NameAndPrice() {
    return View(myProduct);
}
...
```

Затем в представлении используется Razor-выражение `@Model` для получения значений интересуемых свойств:

```
...
The product name is @Model.Name and it costs $@Model.Price
...
```

Строку, предназначенную для отображения, можно было бы создать в методе действия и передать ее представлению как объект модели представления. Это бы сработало, но такой подход разрушает преимущество шаблона MVC и уменьшает возможности по внесению изменений в будущем. Как уже было сказано, мы еще вернемся к этой теме, но вы должны запомнить, что инфраструктура MVC Framework не принуждает правильно применять шаблон MVC и нужно учитывать влияние решений, принимаемых во время проектирования и написания кода.

Вставка значений данных

Самое простое, что можно делать с помощью выражения Razor — это вставка значения данных в разметку. Это осуществляется с использованием выражения `@Model`, позволяющего ссылаться на свойства и методы, которые определены объектом модели представления, или выражения `@ViewBag` для ссылки на свойства, определенные динамически с помощью средства `ViewBag` (рассматриваемого в главе 2).

Вы уже видели примеры обеих выражений, но для полноты мы добавили в контроллер `Home` новый метод действия по имени `DemoExpressions`, который передает данные представлению с применением объекта модели и объекта `ViewBag`. Определение этого нового метода действия приведено в листинге 5.13.

Листинг 5.13. Метод действия DemoExpression

```
using Razor.Models;
using System;
using System.Web.Mvc;

namespace Razor.Controllers {
    public class HomeController : Controller {
        Product myProduct = new Product {
            ProductID = 1,
            Name = "Kayak",
            Description = "A boat for one person",
            Category = "Watersports",
            Price = 275M
        };

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

        public ActionResult NameAndPrice() {
            return View(myProduct);
        }

        public ActionResult DemoExpression() {

            ViewBag.ProductCount = 1;
            ViewBag.ExpressShip = true;
            ViewBag.ApplyDiscount = false;
            ViewBag.Supplier = null;
            return View(myProduct);
        }
    }
}
```

Для демонстрации этих базовых типов выражений мы создали строго типизированное представление DemoExpression.cshtml, контент которого показан в листинге 5.14.

Листинг 5.14. Контент файла представления DemoExpression

```
@model Razor.Models.Product

@{
    ViewBag.Title = "DemoExpression";
}
<table>
    <thead>
        <tr><th>Property</th><th>Value</th></tr>
    </thead>
    <tbody>
        <tr><td>Name</td><td>@Model.Name</td></tr>
        <tr><td>Price</td><td>@Model.Price</td></tr>
        <tr><td>Stock Level</td><td>@ViewBag.ProductCount</td></tr>
    </tbody>
</table>
```

В этом примере создана простая HTML-таблица, ячейки которой заполняются свойствами из объекта модели и объекта ViewBag. На рис. 5.12 можно видеть результат запуска приложения и перехода по URL типа /Home/DemoExpression. Он лишь подтверждает работу базовых выражений Razor, которые применялись в примерах до сих пор.

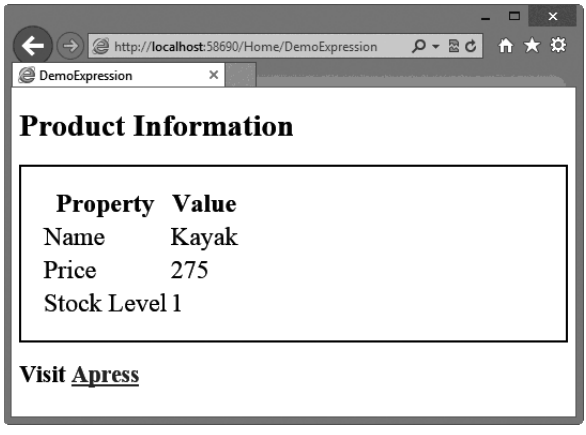

Рис. 5.12. Использование базовых выражений Razor для вставки значений данных в HTML-разметку

Результат нельзя назвать впечатляющим, поскольку никакие CSS-стили к HTML-элементам, сгенерированным представлением и компоновкой, не применялись, однако этот пример предназначен только для иллюстрации способа использования базовых выражений Razor для отображения данных, передаваемых из метода действия в представление.

Установка значений атрибутов

Во всех рассмотренных до сих пор примерах устанавливался контент элементов, но выражения Razor можно также применять и для установки значений атрибутов элемента. В листинге 5.15 показано измененное представление `DemoExpression`, в котором свойства `ViewBag` используются в качестве значений атрибутов. Способ, которым механизм Razor обрабатывает атрибуты в MVC 4, довольно интеллектуален, и это одна из областей, которые были улучшены по сравнению с версией MVC 3.

Листинг 5.15. Использование выражений Razor для установки значений атрибутов

```
@model Razor.Models.Product

@{
    ViewBag.Title = "DemoExpression";
    Layout = "~/Views/_BasicLayout.cshtml";
}

<table>
    <thead>
        <tr><th>Property</th><th>Value</th></tr>
    </thead>
    <tbody>
        <tr><td>Name</td><td>@Model.Name</td></tr>
        <tr><td>Price</td><td>@Model.Price</td></tr>
        <tr><td>Stock Level</td><td>@ViewBag.ProductCount</td></tr>
    </tbody>
</table>

<div data-discount="@ViewBag.ApplyDiscount" data-express="@ViewBag.ExpressShip"
     data-supplier="@ViewBag.Supplier">
    The containing element has data attributes
</div>
```

```
Discount:<input type="checkbox" checked="@ViewBag.ApplyDiscount" />
Express:<input type="checkbox" checked="@ViewBag.ExpressShip" />
Supplier:<input type="checkbox" checked="@ViewBag.Supplier" />
```

Мы начинаем с использования выражений Razor для установки значений ряда атрибутов данных в элементе `div`. Атрибуты данных, имена которых начинаются с `data-`, в течение многих лет были неформальным способом создания специальных атрибутов, а теперь они являются частью формального стандарта HTML5. Для установки значений этих атрибутов применяются свойства `ApplyDiscount`, `ExpressShip` и `Supplier` объекта `ViewBag`. Запустите пример приложения, обратитесь к методу действия и взгляните на HTML-разметку, сгенерированную в результате визуализации страницы. Вы увидите, что механизм Razor установил значения атрибутов:

```
...
<div data-discount="False" data-express="True" data-supplier="">
    The containing element has data attributes
</div>
...
```

Значения `False` и `True` соответствуют булевским значениям свойств в объекте `ViewBag`, но механизм Razor сделал кое-что полезное для свойства со значением `null`, визуализировав для него пустую строку.

Вторая часть разметки, добавленной к представлению, намного более интересна. В ней определен ряд флажков, атрибуты `checked` которых устанавливаются в те же самые свойства объекта `ViewBag`, что и атрибуты данных. Сгенерированная Razor разметка HTML выглядит следующим образом:

```
...
Discount: <input type="checkbox" />
Express: <input type="checkbox" checked="checked" />
Supplier: <input type="checkbox" />
...
```

В MVC 4 механизм Razor осведомлен о принципе использования таких атрибутов, как `checked`, когда конфигурацию элемента изменяет присутствие самого атрибута, а не его значения. Если бы механизм Razor вставил `False`, `null` или пустую строку в качестве значения атрибута `checked`, то отображаемый браузером флажок оказался бы отмеченным. Поэтому в случае значения `false` или `null` механизм Razor полностью удаляет атрибут из элемента, обеспечивая согласованный с данными представления результат, как показано на рис. 5.13.

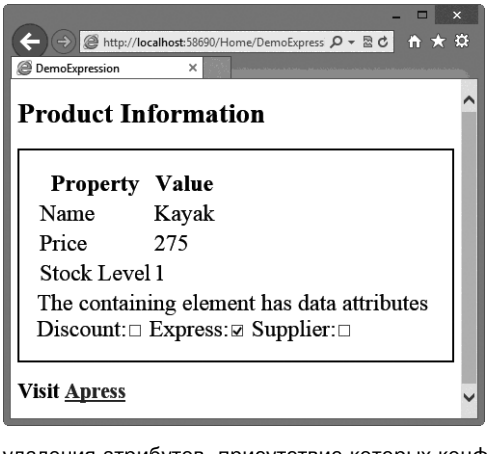

Рис. 5.13. Результат удаления атрибутов, присутствие которых конфигурирует их элемент

Использование условных операторов

Механизм Razor способен обрабатывать условные операторы, а это значит, что можно настраивать вывод представления на основе значений данных представления. Мы начинаем рассматривать наиболее важные части Razor, позволяющие создавать сложные и гибкие компоновки, которые по-прежнему остаются простыми в понимании и сопровождении. В листинге 5.16 показан обновленный контент файла представления `DemoExpression.cshtml`, включающий условный оператор.

Листинг 5.16. Использование условного оператора Razor

```
@model Razor.Models.Product

@{
    ViewBag.Title = "DemoExpression";
    Layout = "~/Views/_BasicLayout.cshtml";
}

<table>
    <thead>
        <tr><th>Property</th><th>Value</th></tr>
    </thead>
    <tbody>
        <tr><td>Name</td><td>@Model.Name</td></tr>
        <tr><td>Price</td><td>@Model.Price</td></tr>
        <tr>
            <td>Stock Level</td>
            <td>
                @switch ((int)ViewBag.ProductCount) {
                    case 0:
                        @: Out of Stock
                        break;
                    case 1:
                        <b>Low Stock (@ViewBag.ProductCount)</b>
                        break;
                    default:
                        @ViewBag.ProductCount
                        break;
                }
            </td>
        </tr>
    </tbody>
</table>
```

Чтобы начать условный оператор, перед условным ключевым словом C# (в этом примере `switch`) помещается символ `@`. Блок кода завершается закрывающей фигурной скобкой (`}`), как в случае обычного блока кода C#.

Совет. Обратите внимание, что для использования в операторе `switch` значение свойства `ViewBag.ProductCount` должно быть приведено к типу `int`. Причина в том, что оператор `switch` работает только со специфическим набором типов и не может оценивать динамическое свойство без приведения подобного рода.

Внутри блока кода Razor можно включать в вывод представления HTML-элементы и значения данных, просто определяя HTML-разметку и выражения Razor, например:

```
...
<b>Low Stock (@ViewBag.ProductCount)</b>
```

```
...
@ViewBag.ProductCount
...
```

Помещать элементы или выражения в кавычки либо отмечать их каким-то специальным образом не требуется — механизм Razor будет интерпретировать это как вывод, который должен обрабатываться обычным образом. Тем не менее, если нужно вставить литеральный текст в представление, когда он не содержится в HTML-элементе, об этом понадобится сообщить Razor, добавив к строке префикс, как показано ниже:

```
...
@: Out of Stock
...
```

Символы `@:` предотвращают обработку механизмом Razor этой строки как оператора C#, что является стандартным поведением в отношении текста. Результат выполнения нашего условного оператора можно видеть на рис. 5.14.

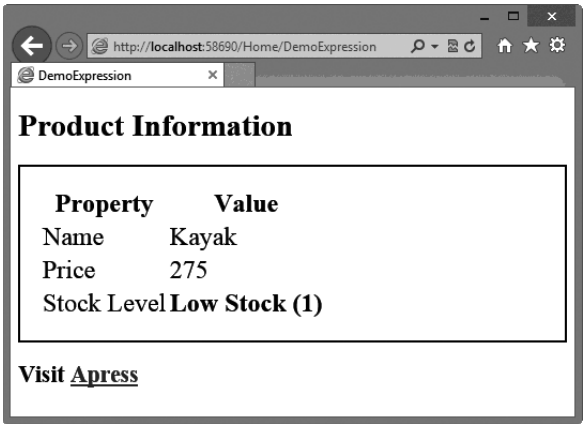

Рис. 5.14. Использование оператора `switch` в представлении Razor

Условные операторы играют важную роль в представлениях Razor, т.к. они позволяют приспосабливать контент к значениям данных, которые представление получает от метода действия. В качестве дополнительной демонстрации в листинге 5.17 приведено представление `DemoExpression.cshtml` с добавленным оператором `if` — еще одним очень распространенным условным оператором.

Листинг 5.17. Использование оператора `if` в представлении Razor

```
@model Razor.Models.Product

@{
    ViewBag.Title = "DemoExpression";
    Layout = "~/Views/_BasicLayout.cshtml";
}

<table>
    <thead>
        <tr><th>Property</th><th>Value</th></tr>
    </thead>
    <tbody>
        <tr><td>Name</td><td>@Model.Name</td></tr>
        <tr><td>Price</td><td>@Model.Price</td></tr>
        <tr>
```

```
            <td>Stock Level</td>
            <td>
               @if (ViewBag.ProductCount == 0) {
                  @:Out of Stock
               } else if (ViewBag.ProductCount == 1) {
                  <b>Low Stock (@ViewBag.ProductCount)</b>
               } else {
                  @ViewBag.ProductCount
               }
            </td>
         </tr>
      </tbody>
   </table>
```

Этот условный оператор выдает те же самые результаты, что и оператор `switch`, но мы просто хотели продемонстрировать применение условных операторов C# в представлениях Razor. Мы объясним, как все это работает, в главе 18, когда начнем рассматривать представления более глубоко.

Перечисление массивов и коллекций

При разработке приложения MVC часто необходимо выполнять перечисление контента массива или другой разновидности коллекции объектов с генерацией подробной информации для каждого объекта. Чтобы продемонстрировать, как это делается, мы определили в контроллере `Home` новый метод действия по имени `DemoArray` (листинг 5.18).

Листинг 5.18. Метод действия `DemoArray`

```
using Razor.Models;
using System;
using System.Collections.Generic;
using System.Web.Mvc;

namespace Razor.Controllers {
   public class HomeController : Controller {
      Product myProduct = new Product {
         ProductID = 1,
         Name = "Kayak",
         Description = "A boat for one person",
         Category = "Watersports",
         Price = 275M
      };

      // ...для краткости другие методы действий не показаны...

      public ActionResult DemoArray() {
         Product[] array = {
            new Product {Name = "Kayak", Price = 275M},
            new Product {Name = "Lifejacket", Price = 48.95M},
            new Product {Name = "Soccer ball", Price = 19.50M},
            new Product {Name = "Corner flag", Price = 34.95M}
         };
         return View(array);
      }
   }
}
```

Этот метод действия создает объект `Product[]`, который содержит простые значения данных, и передает его методу `View` для визуализации с использованием стандартного представления.

При создании представления среда Visual Studio не предлагает варианты для массивов и коллекций, поэтому детали требуемого типа придется вводить вручную в диалоговом окне Add View (Добавление представления). На рис. 5.15 показано, как было создано представление `DemoArray.cshtml` с указанием `Razor.Models.Product[]` в качестве типа модели представления.

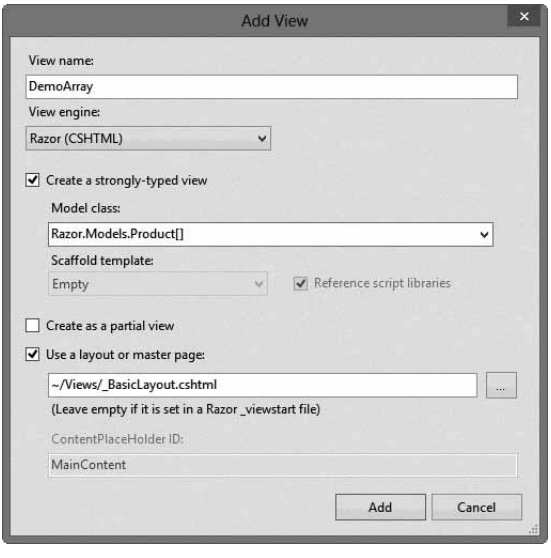

Рис. 5.15. Ручная установка типа модели представления для строго типизированного представления

Контент файла представления `DemoArray.cshtml` приведен в листинге 5.19; он включает добавления, предназначенные для визуализации пользователю деталей элементов массива.

Листинг 5.19. Контент файла `DemoArray.cshtml`

```
@model Razor.Models.Product[]

@{
    ViewBag.Title = "DemoArray";
    Layout = "~/Views/_BasicLayout.cshtml";
}

@if (Model.Length > 0) {
    <table>
        <thead><tr><th>Product</th><th>Price</th></tr></thead>
        <tbody>
            @foreach (Razor.Models.Product p in Model) {
                <tr>
                    <td>@p.Name</td>
                    <td>$@p.Price</td>
                </tr>
            }
```

```
        </tbody>
    </table>
} else {
    <h2>No product data</h2>
}
```

Здесь с помощью оператора `@if` варьируется контент на основе длины обрабатываемого массива, а посредством выражения `@foreach` выполняется перечисление контента массива с генерацией строки HTML-таблицы для каждого элемента массива. Как видите, эти выражения соответствуют своим аналогам из C#; мы создали в цикле `foreach` локальную переменную по имени `p` и ссылаемся на нее с использованием выражений Razor вида `@p.Name` и `@p.Price`.

В результате генерируется элемент `h2`, если массив пуст, и по одной строке HTML-таблицы для каждого элемента массива в противном случае. Поскольку в рассматриваемом примере данные являются статическими, мы всегда получаем одни и те же результаты, которые показаны на рис. 5.16.

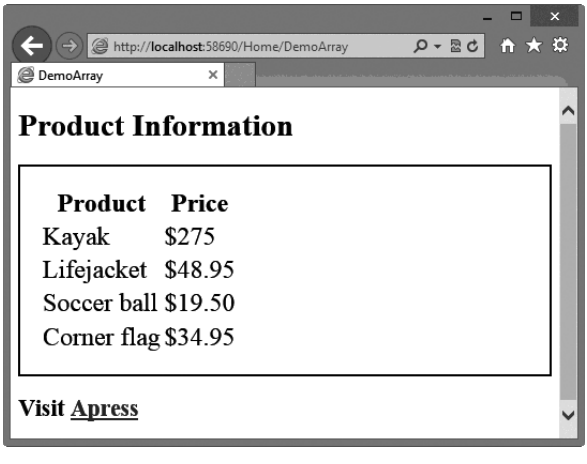

Рис. 5.16. Генерация элементов с применением оператора `foreach`

Работа с пространствами имен

В последнем примере вы наверняка заметили, что для ссылки на `Product` в цикле `foreach` используется полностью определенное имя:

```
...
@foreach (Razor.Models.Product p in Model) {
...
```

Это может стать утомительным в сложных представлениях, содержащих множество ссылок на модель представления и другие классы. Привести в порядок представления можно за счет применения выражения `@using`, чтобы обеспечить для представления контекст определенного пространства имен, как это делается для обычного класса C#. В листинге 5.20 демонстрируется применение выражения `@using` к ранее созданному представлению `DemoArray.cshtml`.

Листинг 5.20. Применение выражения @using

```
@using Razor.Models
@model Product[]

@{
    ViewBag.Title = "DemoArray";
    Layout = "~/Views/_BasicLayout.cshtml";
}

@if (Model.Length > 0) {
    <table>
        <thead><tr><th>Product</th><th>Price</th></tr></thead>
        <tbody>
            @foreach (Product p in Model) {
                <tr>
                    <td>@p.Name</td>
                    <td>$@p.Price</td>
                </tr>
            }
        </tbody>
    </table>
} else {
    <h2>No product data</h2>
}
```

Представление может содержать множество выражений `@using`. В показанном выше коде выражение `@using` используется для импорта пространства имен `Razor.Models`, что позволяет не указывать пространство имен в выражении `@model` и внутри цикла `foreach`.

Резюме

В этой главе был предложен обзор механизма визуализации Razor и показано, как его можно применять для генерации HTML-контента. Мы объяснили, каким образом ссылаться на данные, переданные из контроллера, через объект модели представления и объект `ViewBag`, а также продемонстрировали использование выражений Razor для настройки ответа пользователю на основе данных, с которыми производится работа. В оставшихся главах книги будет приведено много разных примеров применения Razor, а в главе 18 детально рассматривается функционирование самого механизма представлений MVC. В следующей главе мы опишем важные инструменты для разработки и тестирования, лежащие в основе MVC Framework, которые помогут строить максимально эффективные проекты.